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. 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

Contents

Chapter 1

Abstract

Chapter 2 gives a literature overview on “reachability” and “LCR”. Also we discuss some of the ideas for actual algorithms. Chapter 3 discusses the experimental set-up used in the experiment and discusses the datasets. Chapter 4 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 5 shows the results of the experiments. Chapter 6 concludes on all material and provides work that can be done in the future.

Chapter 2

Problem statement

2.0.1 Reachability in graphs

The use of reachability queries in graphs has been studied extensively [9, Chapter 6] [8, 1]. Answering these kinds of queries is ubiquitous in many applications. The "reachability problem" and a "path" can formally be defined as:

Path: Let $G = (V, E)$ be a (directed) graph with $|V| = n$ and $|E| = m$. Let $s, t \in V$ be two arbitrary nodes. A path P exists in G from s to t if and only if: $P = \langle s, w_0, \ldots, w_k, t\rangle$ where $k \geq 0$ and $\forall[1 \leq i \leq k | w_i \in V \wedge (w_{i-1}, w_i) \in E] \wedge w_0 \in V \wedge (s, w_0) \in E \wedge (w_k, t) \in E$ or $P = \langle s, t\rangle \wedge (s, t) \in E$ or $P = \langle s\rangle \wedge s = t$.

Reachability: Let $G = (V, E)$ be a (directed) graph with $|V| = n$ and $|E| = m$. Let $s, t \in V$ be two arbitrary nodes. If there exists a path from s to t, i.e. $s \rightarrow t$, we say t is reachable from s.

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]. Answering these queries without an index by running a depth- or breadth-first search live which has a running time of $O(n+m)$ but has zero construction time and index size. For large graphs both methods are unacceptable. For example, storing a transitive closure as a bit-matrix for $n = 1,000,000$ requires 125GB. We need methods that have lower storage requirements at the expense of a higher query time.

2.0.2 Label-constrained reachability

In this thesis, we describe and make a survey of the problem of label constrained reachability (LCR) and regular path queries. In the LCR-case edges have a label from a pre-defined label set. Formally this can be defined as:

LCR: Let $G = (V, E, \mathcal{L})$ be a (directed) graph with $|V| = n$ and $|E| = m$. $\mathcal{L}$ is a set of labels and $E \subseteq V \times V \times \mathcal{L}$. Let $s, t \in V$ be two arbitrary nodes. Also let $L \subseteq \mathcal{L}$ be a subset of the labels. We say that there is a L-path from s to t if there exists a path from s to t using only edges with labels in L. When there is a L-path from s to t, we write $s \overset{L}{\rightsquigarrow} t$.

An example of a directed labelled graph can be seen in Figure 2.1. There exists a $\{a, b\}$-path from 1 to 5. 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

[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

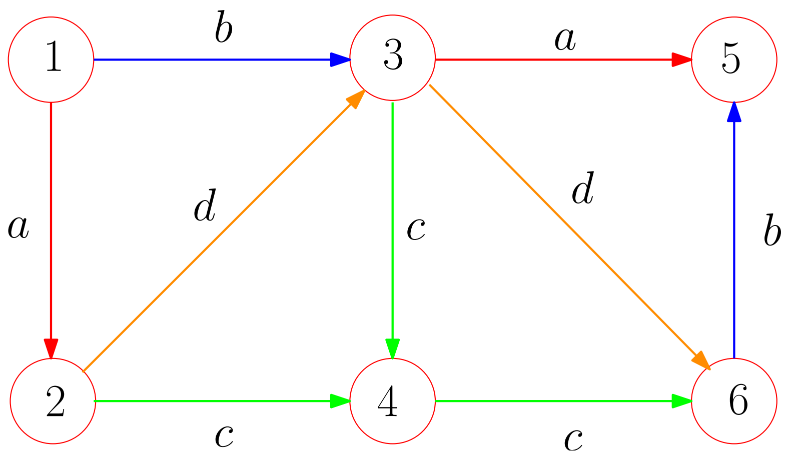


Figure 2.1: An example of a graph with labels. From node 1 to 6 there are for instance an $\{a, b\}$-path, an $\{a, c\}$-path, an $\{a, d\}$-path, a $\{b, c\}$-path and a $\{b, d\}$-path.

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.

Formally a regular path query (also referred to as LCR-query or simply query in the remainder of our thesis) can be defined as:

LCR-query: Let $s, t \in V$ and $\mathcal{L} \subset L$. Then, we can define a query $q = (s, t, \mathcal{L})$. If $s \overset{L}{\rightsquigarrow} t$, then q is said to be true. Otherwise, q is said to be false.

An example query on the graph in Figure 2.1 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 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.

2.0.3 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 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.
In Figure 2.2 we can a simple taxonomy of different approaches. The red dot indicates BFS with its maximal query time and minimal construction time and the blue dot indicates any Exact-method. The exact shape of the curve is something that needs to be investigated.

[4]http://www.cnet.com/news/googles-knowledge-graph-tripled-in-size-in-seven-months/

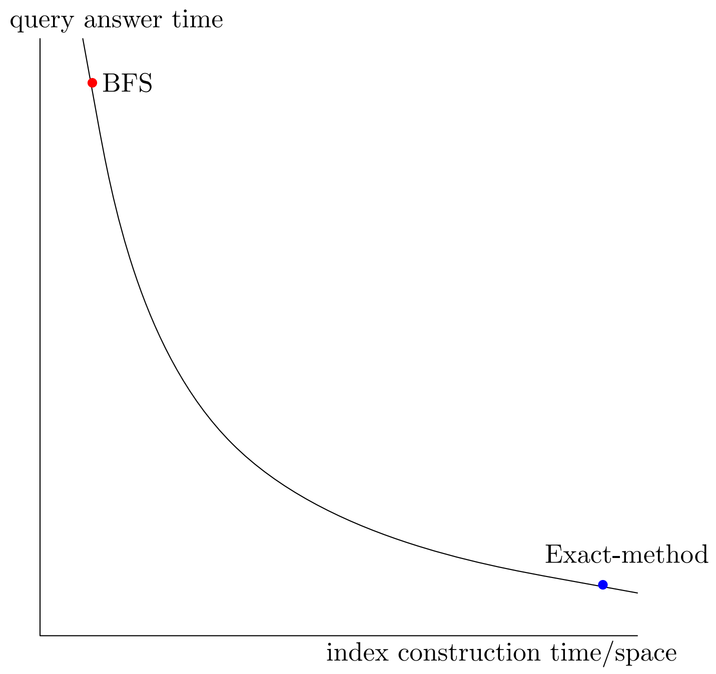


Figure 2.2: This chart shows how BFS (red dot) and any Exact-method (blue dot) relate to one another. Other approaches will likely lie on the curve between the red and the blue dot. The shape of the curve can be different from the figure.

2.0.4 Definitions

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

- 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 $ANCS(v) = \{w \in V | w \text{ can reach } v\}$ and the set of descendants $DESC(v) = \{w \in V | v \text{ can reach } w\}$. Note that $v \in ANCS(v) \wedge v \in DESC(v)$.
- In a directed graph, a strongly connected component (SCC) is a subset of the nodes $A \subseteq V$ s.t. $\forall [v, v' \in A | v \rightsquigarrow v']$. A weakly connected component (WCC) is a subset A s.t. $\forall [v, v' \in A | v \rightsquigarrow v' \vee v' \rightsquigarrow v]$.
- In a directed graph, let a path P from $s \in V$ and $t \in V$ be a sequence of the form $\langle s, ..., t \rangle$ s.t. for each pair of consecutive elements in the sequence there is an edge $e \in E$. Let $P[i]$ be the i'th element in a path. Let $\#P$ be the length of the path.
- 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 $P(s,t) = \emptyset$.
- Let $G = (V, E, \mathcal{L})$ be a labelled (directed) graph with $|V| = n$ and $|E| = m$. $\mathcal{L}$ 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 $l \in \mathcal{L}$. Also let $Label(e)$ be a mapping from edge $e \in E$ to its corresponding label.
- In a labelled (directed) graph, let $Labels(P)$ be defined as: $\bigcup_{i=1}^{\#P-1} Label(P[i], P[i+1])$. This is also called "the label of a path".
- In a labelled (directed) graph let $P_{\min}(s,t)$ be the set of simple paths s.t. for any two paths $P, P' \in P(s,t)$ we have that $Labels(P) \not\subseteq Labels(P') \vee Labels(P') \not\subseteq Labels(P)$. Any path in this set is said to be 'minimal', as is $P_{\min}(s,t)$ by itself.

Chapter 3

Literature analysis

Techniques that are used for determining reachability in normal graphs could be a starting point for a technique to solve reachability in labeled graphs or even improve existing techniques.

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. 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 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 two lists $In(v)$ and $Out(v)$. These lists contain (a subset of) the set of ancestors of v and the set of descendants of v. The query $u \rightsquigarrow v$ then evaluates to true if and only if:

$$In(v) \cap Out(u) \neq \emptyset$$

The goal is to find a minimal 2-hop cover. However finding such a cover is NP-hard [9, Chapter 6]. Cohen et al. [3] gives an approximation algorithm which gives a cover at most $O(\log(n))$ larger. The worst case query time is $O(m^{1/2})$ and the index size is $O(nm^{1/2})$. In Figure 3.1, we can see a simple graph. A full-hop cover and a

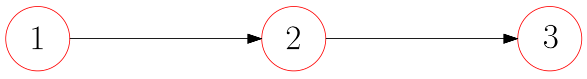


Figure 3.1: A simple graph. In the table below, one can see a full hop-cover versus a minimal one.

minimal cover can be seen in the Table 3.1 below.

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

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

3.1.1 Cohen's 2-hop cover approximation

In Algorithm 1 we show an algorithm that computes a 2-hop cover [3]. Let $TC(G)$ be the edge transitive closure of G. $(u, v) \in TC(G)$ implies that u can reach v. Consider a node w. Let $A_w \subseteq ANCS(w)$ and $D_w \subseteq DECS(w)$. 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 $|C_w| = |A_w| + |D_w|$. The algorithm repeatedly looks for a cluster C_w that has a maximal coverage.

Algorithm 1 2Hop-Cover(G)

```
1: compute $TC(G)$
2: $TC'(G) \leftarrow TC(G)$
3: $L_{in}(v), L_{out}(v) \leftarrow \emptyset$ for all $v \in V$
4: while $TC' \neq \emptyset$ do
5:    find $\max_{w\in} |C_w \cap TC'(G)|/(|A_w| + |D_w|)$
6:    for $u \in A_w \wedge v \in D_w$ do
7:       remove $(u, v)$ from $TC'(G)$
8:       add $w$ to $L_{in}(v)$
9:       add $w$ to $L_{out}(u)$
10:   end for
11: end while
```

All pairs that w covers are removed from TC' and added to $L_{in}(v)$ and $L_{out}(u)$. An interesting thing to note 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. One can see that the memory requirements of this algorithm are really large.

3.1.2 Applying 2-hop cover to LCR

For applying a 2-hop cover in the LCR-case we would first need to compute a 2-hop cover ignoring the edge labels and subsequently for any pair (v, w) where $w \in Out(v) \vee v \in In(w)$ we need to find all paths between v and w. Essentially, we need to compute $P(v, w)$. This could be cumbersome if this needs to be repeated for any pair (v, w). One can imagine more efficient strategies.

3.2 Graph partitioning

For large graphs I/O-efficiency is often a serious concern. Graph-partitioning approaches [9, Chapter 6] can address this issue. The main idea is to divide a graph G into k subgraphs $G_1, \ldots, G_k$. Each node $v \in V$ is in exactly one of the subgraphs. An edge can be either "internal" which means that its endpoints are in the same subgraph, or "external" which means that its endpoints are in two different subgraphs. Each subgraph should be of a size that fits into memory.

Reachability could be answered in this setting by creating a "skeleton graph" $G' = (V', E')$. This graph gives a global view of the original graph in which $V' = \bigcup_{i=1}^{k} G_i$ and $E' = E_C \cup E_S$ where E_C is the set of external edges and E_S is a set of edges (v_i, w_i) with $v_i, w_i \in V_i$ whenever $v_i \rightsquigarrow w_i$. Reachability within a subgraph and within the skeleton graph can be combined to answer reachability queries on the full graph.

3.2.1 Exact-method: ClusteredExact

While developing algorithms for the index, we found that the index construction time was one of the main bottlenecks. Hence, we got the idea to not store or compute the full index. Rather we divide the graph into $\lfloor n/k \rfloor$ clusters and make a high-level graph G' for the clusters. In Figure 3.2 the idea can be seen. Each node is either external or internal to the cluster which means whether or not the node has connections to other clusters or not. We try to keep the number of external nodes as low as possible. For each cluster we compute a full Exact-index possibly using the method for this in the previous section. For each external node in a cluster we keep a list of all nodes it connects to. This is a much smaller index. The Exact-index for a graph with n nodes would be of order $O(dn^2)$ with $d \leq 2^{|\mathcal{L}|}$. The total size of the indices of all clusters would be of order $O(k \cdot c(\lfloor n/k \rfloor \cdot \lfloor n/k \rfloor))$ which is $O(cn^2/k)$ with $c \leq 2^{|\mathcal{L}|}$. We don't know whether $d \geq c$ or $c \geq d$. One can argue

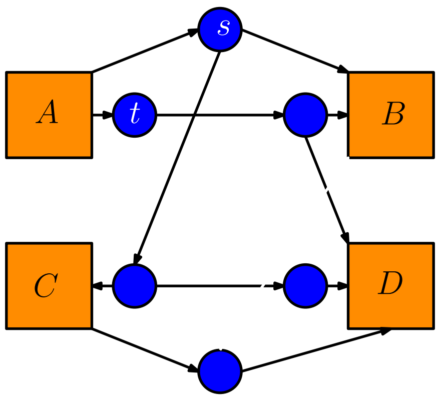


Figure 3.2: A high-level graph. The orange boxes are all graphs (or 'clusters'). The blue nodes are the entry and exit points of the clusters. For the orange boxes an Exact index is computed for all internal nodes. For the blue nodes we maintain which clusters they connect to. A query (s, t, L) can be answered by combining both sets of information.

that for larger clusters there will be more paths between any v and w and that this increases the probability of a path with very few labels which would prune many entries from the index. One can also argue that having fewer paths between a pair (v, w) means fewer label sets between v and w.

To determine reachability from s to t using L we first look up the clusters of s and t, then look through which paths s and t can be connected using G'. For each cluster on a path from s to t we verify whether we can cross that cluster using labels in L.

3.3 Graph updates

The graph can be updated in four different ways: adding or removing a new node or adding or removing an edge. For labeled 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} that consists of all ancestors u of v and all descendants of u. Let $G_{REL} = (V_{REL}, E_{REL})$. For this part, a 2-hop cover L' is computed. Next, we look at all connections $(a, d) \in E$ of which $a \in V_{REL} \lor d \in V_{REL}$. If $a \in V_{REL}$, then $d \in V_{REL}$ and $L_{out}(a) = L'_{out}(a)$ and $L_{in}(d) = L'_{in}(d)$. If $d \notin V_{REL}$, then $(L_{in}(d) \setminus V_{REL}) \cup L'_{in}(d)$. 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 nodes on paths from a to d. Obviously $v \in W$. The trade-off between computing time and storage can be found in storing all nodes $w \in W$ in $L_{out}(a)$ and $L_{in}(d)$. If node $v \in W$ gets deleted, we can safely delete v in all $L_{out}(a)$ and $L_{in}(d)$ because there is another route. Of course, this method leads to a much lower compression rate.

3.4 Bonchi et al.

Bonchi et al. [1] tackle the problem of finding the minimal distance in labeled undirected graphs between two nodes, which is an extension of "LCR". 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.

Both approaches build on the definition of SP-minimality. 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 $(u, x), u \in V \land x \in X$ 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 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 $x \in X$ and $v \in V$ for a set of candidate label sets. The brute force method takes $O(2^{|L|}k(m + n|L|))$.

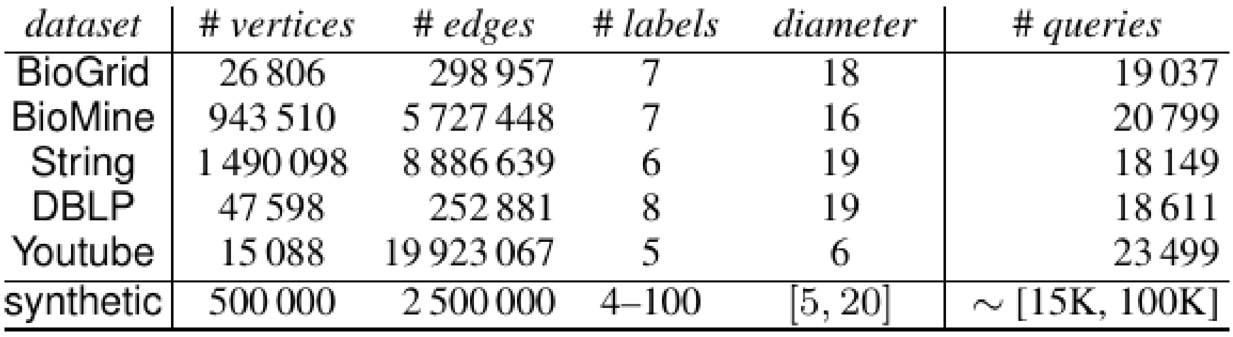
The second method assigns a single 'color' to each landmark $x \in X$. For each node $v \in V$ we store the "mono-chromatic" distance to each landmark. The "mono-chromatic" distance is obtained by only using edges of a single color. The distances between landmarks are computed using a "bi-chromatic" distance metric, that is using two types of labels. 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, 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 $d_C(x, u) \geq |C|$ because otherwise at least one label $c \in C$ 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.

3.4.1 Results of Bonchi et al.

In this section we discuss the results of Bonchi et al. The code was implemented in Java and ran on a single core of an Intel Xeon Server 2.83Ghz with 64 GB RAM.

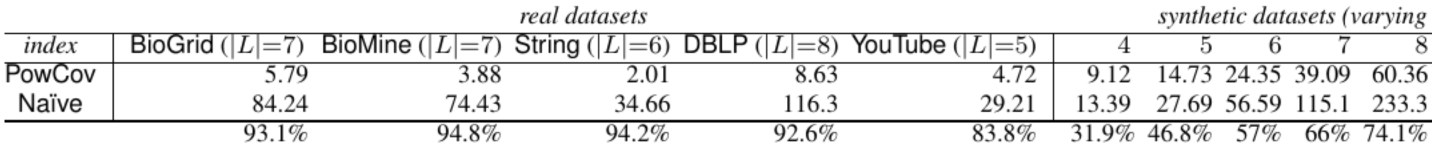
In Figure 3.3 we can see the main characteristics of the datasets that were used by Bonchi et al. The last line shows a synthetic dataset that was created by a generator algorithm. In Figure 3.3 we can see the index



dataset	*# vertices*	*# edges*	*# labels*	*diameter*	*# queries*
BioGrid	26 806	298 957	7	18	19 037
BioMine	943 510	5 727 448	7	16	20 799
String	1 490 098	8 886 639	6	19	18 149
DBLP	47 598	252 881	8	19	18 611
Youtube	15 088	19 923 067	5	6	23 499
synthetic	500 000	2 500 000	4–100	[5, 20]	$\sim$ [15K, 100K]

Figure 3.3: Characteristics of the datasets that were used for the evaluation of the paper by Bonchi et al.

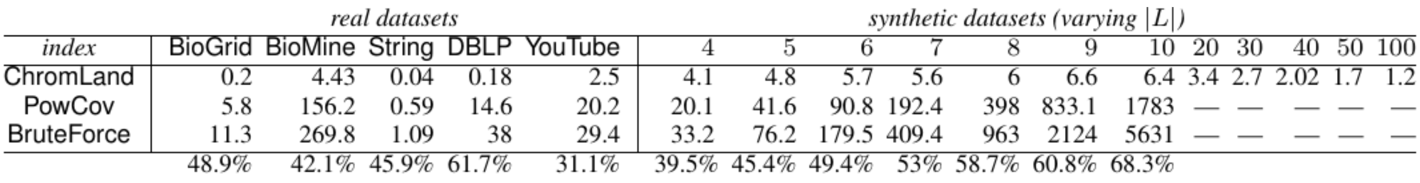
size expressed as the average number of distances stored per vertex landmark pair. The bottom line shows the percentual improvement between Naive and PowCov. The difference between the two is that the Naive approach does not use pruning and hence stores much more pairs. The improvement is quite significant in all cases. The index size gets much larger when L increases.



	real datasets					*synthetic datasets (varying*				
index	BioGrid ($\|L\|$=7)	BioMine ($\|L\|$=7)	String ($\|L\|$=6)	DBLP ($\|L\|$=8)	YouTube ($\|L\|$=5)	4	5	6	7	8
PowCov	5.79	3.88	2.01	8.63	4.72	9.12	14.73	24.35	39.09	60.36
Naïve	84.24	74.43	34.66	116.3	29.21	13.39	27.69	56.59	115.1	233.3
	93.1%	94.8%	94.2%	92.6%	83.8%	31.9%	46.8%	57%	66%	74.1%

Figure 3.4: Index size: the average number of distances stored per vertex landmark pair. The bottom line shows the percentual improvement between Naive and PowCov. Naive does not use pruning.

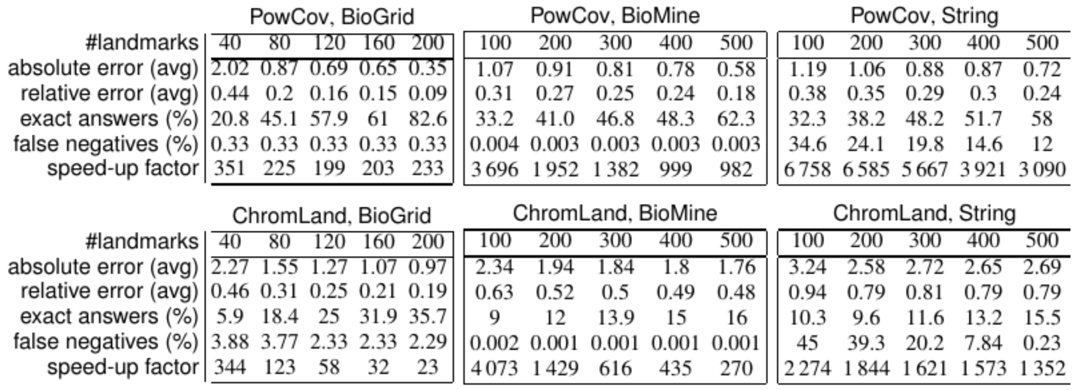
In Figure 3.5 we can see the index construction times (s) of the ChromLand (second algorithm in the previous section), PowCov (first algorithm with pruning) and BruteForce (first algorithm without pruning) per landmark. The results show that the speed-up can be around 70%. We can also see that constructing a full index may take a lot of time, e.g. 20.2 seconds per landmark for **Youtube**. In Figure 3.6 we see the query processing results on real data (**BioGrid**, **BioMine** and **String** datasets) with varying the number of landmarks. For both PowCov and ChromLand the table reports: *i* the absolute and relative errors with respect to the real distances, averaged over all queries evaluated, *ii* the fraction of queries that return the exact answer, *iii* the fraction of queries for



	real datasets					*synthetic datasets (varying $\|L\|$)*												
index	BioGrid	BioMine	String	DBLP	YouTube	4	5	6	7	8	9	10	20	30	40	50	100
ChromLand	0.2	4.43	0.04	0.18	2.5	4.1	4.8	5.7	5.6	6	6.6	6.4	3.4	2.7	2.02	1.7	1.2
PowCov	5.8	156.2	0.59	14.6	20.2	20.1	41.6	90.8	192.4	398	833.1	1783	—	—	—	—	—
BruteForce	11.3	269.8	1.09	38	29.4	33.2	76.2	179.5	409.4	963	2124	5631	—	—	—	—	—
	48.9%	42.1%	45.9%	61.7%	31.1%	39.5%	45.4%	49.4%	53%	58.7%	60.8%	68.3%					

Figure 3.5: Index construction times (s) per landmark

which an infinite distance is returned mistakenly (false negative) *iv* the speed-up factor with respect to shortest path distance computation, e.g. Dijkstra which only considers labels of query set, averaged over all queries. The



	PowCov, BioGrid				
#landmarks	40	80	120	160	200
absolute error (avg)	2.02	0.87	0.69	0.65	0.35
relative error (avg)	0.44	0.2	0.16	0.15	0.09
exact answers (%)	20.8	45.1	57.9	61	82.6
false negatives (%)	0.33	0.33	0.33	0.33	0.33
speed-up factor	351	225	199	203	233

	PowCov, BioMine				
#landmarks	100	200	300	400	500
absolute error (avg)	1.07	0.91	0.81	0.78	0.58
relative error (avg)	0.31	0.27	0.25	0.24	0.18
exact answers (%)	33.2	41.0	46.8	48.3	62.3
false negatives (%)	0.004	0.003	0.003	0.003	0.003
speed-up factor	3 696	1 952	1 382	999	982

	PowCov, String				
#landmarks	100	200	300	400	500
absolute error (avg)	1.19	1.06	0.88	0.87	0.72
relative error (avg)	0.38	0.35	0.29	0.3	0.24
exact answers (%)	32.3	38.2	48.2	51.7	58
false negatives (%)	34.6	24.1	19.8	14.6	12
speed-up factor	6 758	6 585	5 667	3 921	3 090

	ChromLand, BioGrid				
#landmarks	40	80	120	160	200
absolute error (avg)	2.27	1.55	1.27	1.07	0.97
relative error (avg)	0.46	0.31	0.25	0.21	0.19
exact answers (%)	5.9	18.4	25	31.9	35.7
false negatives (%)	3.88	3.77	2.33	2.33	2.29
speed-up factor	344	123	58	32	23

	ChromLand, BioMine				
#landmarks	100	200	300	400	500
absolute error (avg)	2.34	1.94	1.84	1.8	1.76
relative error (avg)	0.63	0.52	0.5	0.49	0.48
exact answers (%)	9	12	13.9	15	16
false negatives (%)	0.002	0.001	0.001	0.001	0.001
speed-up factor	4 073	1 429	616	435	270

	ChromLand, String				
#landmarks	100	200	300	400	500
absolute error (avg)	3.24	2.58	2.72	2.65	2.69
relative error (avg)	0.94	0.79	0.81	0.79	0.79
exact answers (%)	10.3	9.6	11.6	13.2	15.5
false negatives (%)	45	39.3	20.2	7.84	0.23
speed-up factor	2 274	1 844	1 621	1 573	1 352

Figure 3.6: Query processing results with varying the number of landmarks

speed-ups displayed are decent (often ≥ 100), but the false negative rate is often also high 30%.

3.4.2 Applicability to LCR

The idea of building landmarks in a graph can be very useful to LCR, as we do not wish to build a full index that can answer all queries directly but do want to have some index. However, the main difference with the problem of Bonchi et al. [1] is that we have to return either True or False to a query and cannot give an estimate or allow for false/true negatives. Allowing for a certain probability of True- or False-queries could be an extension.

3.5 Zou et al.

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

3.5.1 Zou method

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 $\lambda : E \rightarrow \Sigma$.
- 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 from u. The heap is filled with tuples of the form $(L(p), p, w)$. The entries are sorted along the distance of path p. 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.

Initially, the heap H is filled with all neighbour-tuples of u. Then we iterate over the heap taking an entry 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. From the neighbours of w a set of new triples (neighbour triples) is generated. The neighbour triples must not have a cycle in their path and not be covered already by some entry in H. Only then, they are added to H.

In Figure 3.7 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 neighbour-tuples $(\{b\}, \{u\}, v)$ and $(\{a\}, \{u\}, w)$. These would be added to RS and $M_G(u, -)$ because they are not covered by any entry in RS. Next we would generate for instance $(\{a, b\}, \{u, w\}, v)$ because it is a neighbour tuple of w. However, $(\{b\}, \{u\}, v)$ in RS covers this tuple and hence we do not add this to our index and process the next entry on the heap. The running time of the algorithm is $O(D^d)$ where D is the

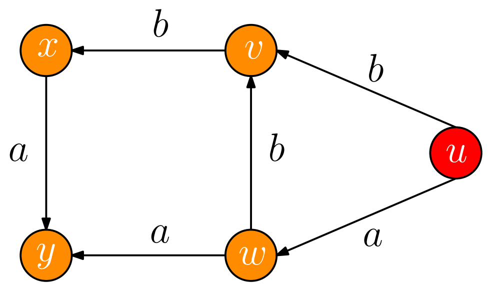


Figure 3.7

maximal out-degree and d is the diameter of the graph. This is the theoretical maximum number of paths from any node $u \in V$.

Creating a DAG

In normal reachability the goal is to find all SCC's in a directed graph. We assume the classical definition of a SCC is used here. Although from Figure 3.8 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 $l \in \mathcal{L}$. After a set of SCC's C has been found, we can get to the next step. 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. In Figure 3.8, we can say 6 is an internal node and 3 is both an in- and out-portal. This results in a bi-partite graph B_i where for every in- and out-portal pair there is an edge either with the label belonging to C_i or no label in case the in- and out-portal have the same vertex id. By concatenating all bipartite graphs and trivial SCC's (those consisting of one node) we get a new DAG D. What the contents of D are precisely is not given in the paper, for instance by giving an example or a definition.

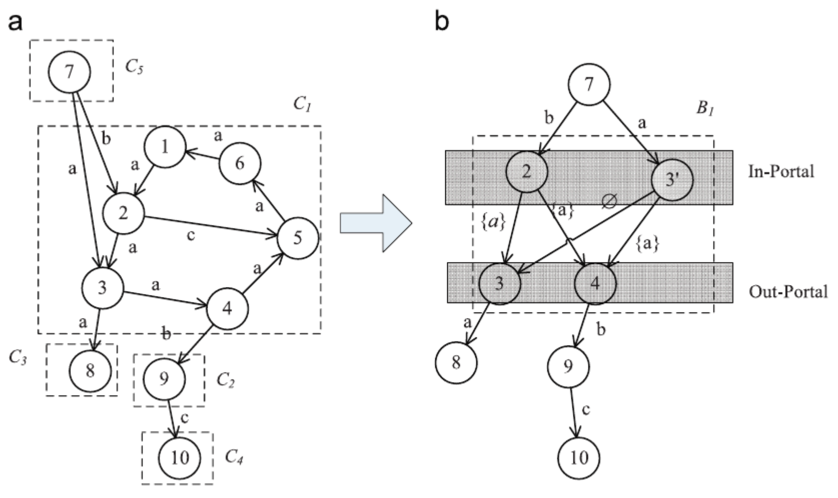


Figure 3.8: Figure 4 from paper [10]. It shows a labeled graph G on the left and the 'classical' SCC's on the right.

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 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 3.9 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.

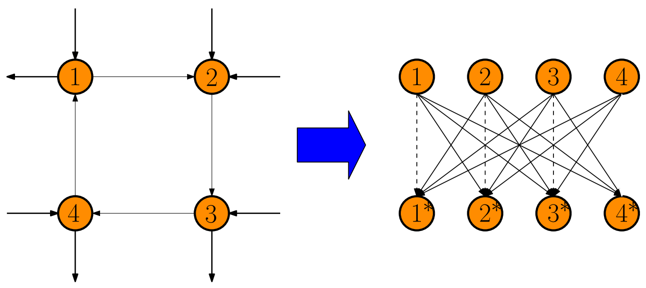


Figure 3.9: A part of a graph eligible to become a SCC on the left. Nodes 1 to 4 could be connected through the same label, e.g. $\{a\}$. On the right, we see that the new number of edges is much higher than in the original. A dashed edge is an edge with no label essentially. This is because 1 and $1'$ represent the same vertex.

Actual algorithm

First we generate $M_{C_i}(u, -)$. This is the internal transitive closure of a strongly connected component. Next, the nodes in D are ordered according to a reverse topological order RT. Note that nodes that are internal to a SCC are not in D.

For $d \in RT$ and $u \in out(d)$ we do $\bigcup_{u \in out(d)} Prune(\lambda(du) \odot M_G(u, -))$. $Prune(S)$ is an operator that removes any path p' that is covered by another path p in the set S, essentially creating a minimal set. $\odot$ concatenates an edge label with a set of paths and their associated label sets. This is done until all nodes $d \in D$ have been traversed.

Figure 3.10 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 $e \in C_3$ after having computed its internal transitive closure. As e has no children, we go to C_2. For each out-portal in C_2 (which is d') connected to an in-portal in C_3 (which is e) 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 this case, we set $M_G(d', -) = Prune(M_G(e, -) \odot \{b\})$ (assuming that the blue edge label is $\{b\}$). For each in-portal i of C_2 we look at all the other out-portals o in C_2. Next, we set $M_G(i, -)$ to $Prune(M_G(o, -), L)$ where L is a label set connecting i and o. In the case of Figure 3.10 we see that in-portal c connects to out-portal d' over label sets $\{a\}$ and $\{b\}$. After we have completed this process for C_2 we do the same for C_1. However now we have only covered the vertices $v \in D \subseteq V$, i.e we have not covered the internal

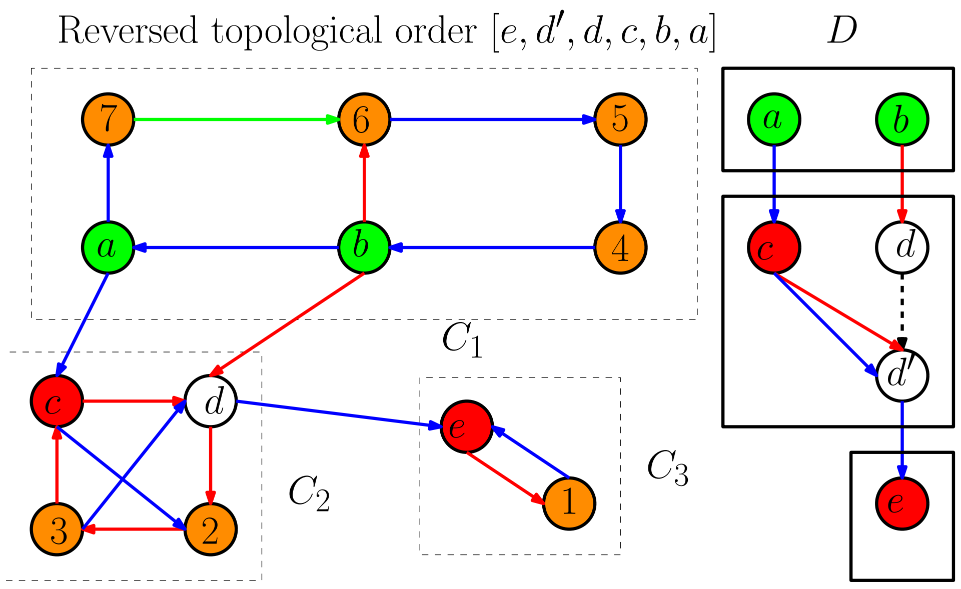


Figure 3.10: On the left a graph with SCC's C_1, C_2 and C_3, which could also be represented by their related bipartite graphs. The edge labels are indicated by the color. A dashed line indicates an edge with no label, which happens when two nodes in D represent the same vertex in G. Inner vertices 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.

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, e.g. in the case of Figure 3.10 1 comes before 3 and 2 and 3 comes before 4. For each inner vertex $v \in PT$ and out-portal d in SCC C' we set $M_G(v, -)$ to $\bigcup_{d \in outp_{C'}} (Prune(M_{C'}(v, d) \odot M_G(d, -))$, where $outp_{C'}$ is the set of out-portals of C' and $M_{C'}(u, d)$ is the minimal set of label sets that connects u and d.

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 5000 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 3.5.1. 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.

3.5.2 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 3.11, 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 3.12 we see the effects of increasing the

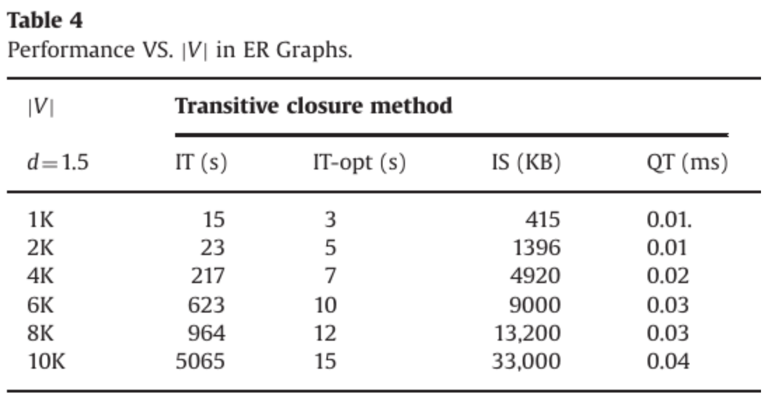


Table 4
Performance VS. $|V|$ in ER Graphs.

$\|V\|$	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 3.11: 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.

density in ER graphs. The number of vertices has been fixed to $|V| = 10,000$.

In Figure 3.13 we see the results for SF-graphs. There is a major difference here compared to 3.11. 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 almost any index, even for 1k graphs, would exceed that much. In Figure 3.11 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 3.12 with a higher degree yield much smaller indices, e.g. for $d = 5$ we see a 186KB index. We find this very strange.

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

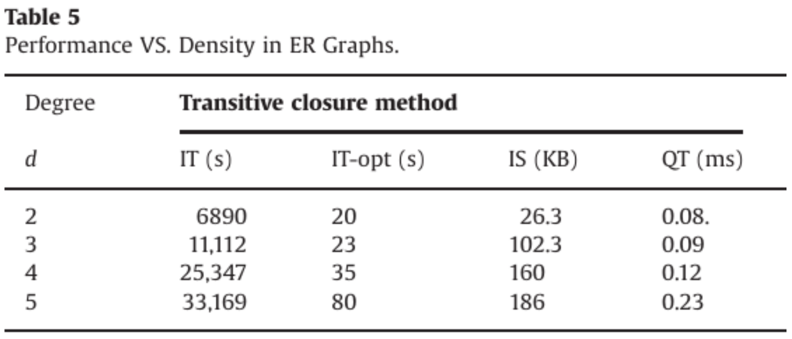


Table 5
Performance VS. Density in ER Graphs.

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 3.12: 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.

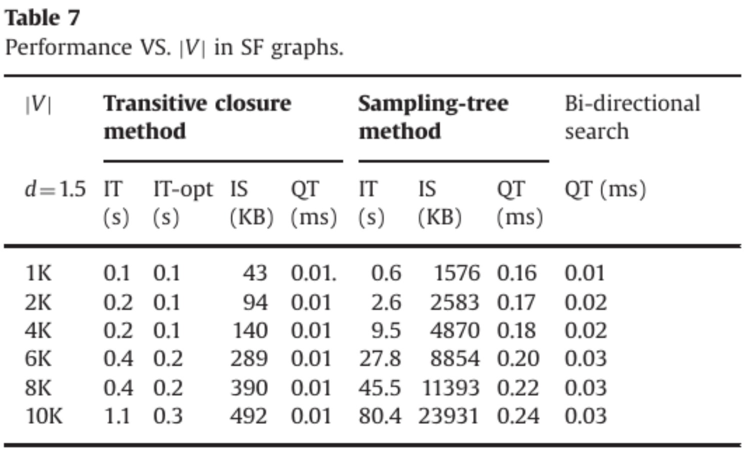


Table 7
Performance VS. $|V|$ in SF graphs.

$\|V\|$	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 3.13: 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.

3.6 Fletcher et al.

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

3.6.1 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 $\text{Out}(v)$, $\text{In}(v)$. We define $\text{In}(v) = \{(s, L) | \exists[P \in P_{min}(v, s) | Labels(P) = L]\}$ and $\text{Out}(v) = \{(s, L) | \exists[P \in P_{min}(s, v) | Labels(P) = L]\}$. A same idea as in the 2-hop cover framework applies here. The formula to determine reachability query (s, t, L) in the labeled-graph is:

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

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.

Table 3.2: This table shows the first and only round of running NeighbourExchange on the graph in Figure 3.14. 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	$(1, \{a, c\}), (2, \{c\}), (2, \{a, b\}), (3, \{a\})$	$\emptyset$

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 $\text{Out}(v)$ or $\text{In}(v)$ respectively.

The second algorithm NeighbourExchange keeps looping as long as there is a node v which had a change in its $\text{In}(v)$ or $\text{Out}(v)$. During each such iteration we loop over all v_i with $1 \le i \le n$. 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.

In Figure 3.14 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. Both

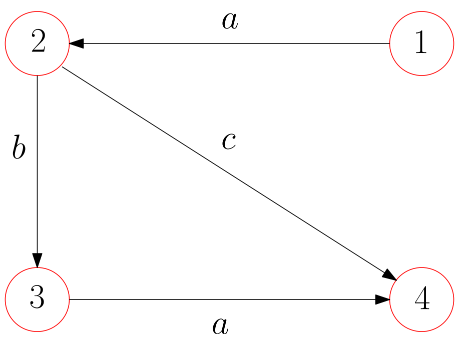


Figure 3.14: A graph with labels. In Table 3.2 we see the updates during the first round of the algorithm NeighbourExchange.

algorithms can prune by not storing or not continuing a search when we add a label (v_i, L) to $\text{In}(v)$ or $\text{Out}(v)$, when there exists a pair $(v_i, L') \in \text{In}(v) \vee (v_i, L') \in \text{Out}(v)$ 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 DoubleBFS do a lot of redundant computation. Suppose we have a graph like in Figure 3.15 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 $\text{Out}(A) = \{B, C, D\}$ and then re-use this result any time we hit vertex A.

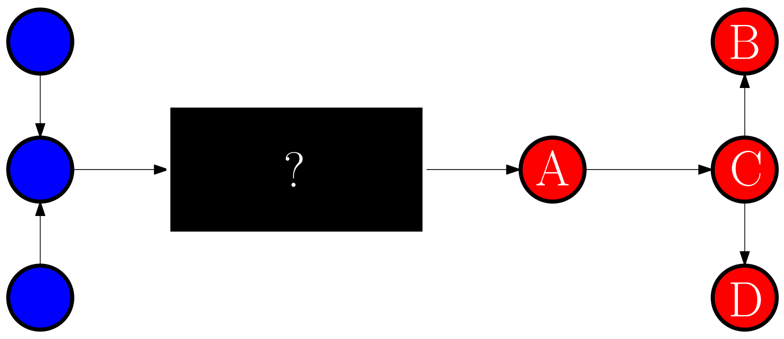


Figure 3.15: A graph. The black box could be any graph which makes the red nodes reachable from the blue nodes.

3.7 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 et al. [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.

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 or 125k), 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.

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 Figures **??** and 4.2, 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 weights.
- **yagoFacts-small** has been taken from ?

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

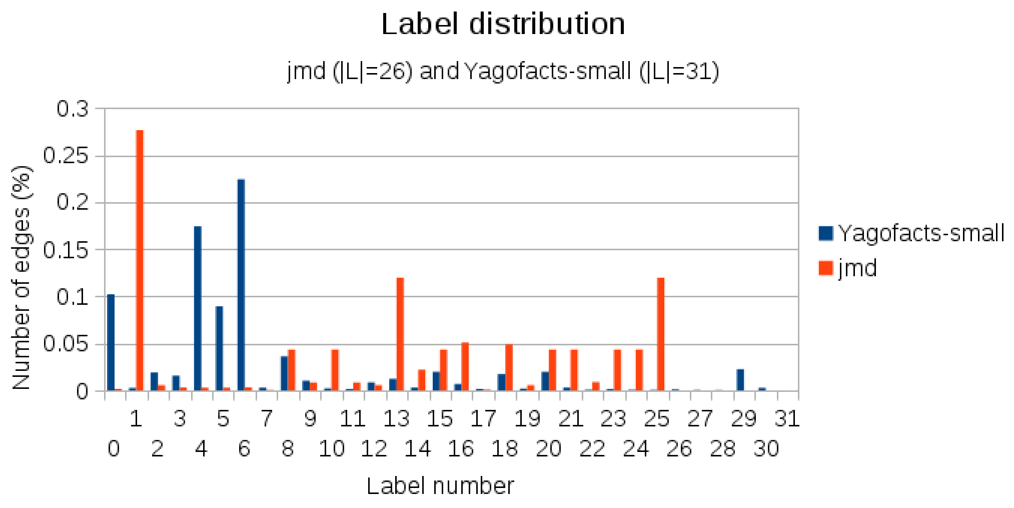


Figure 4.1

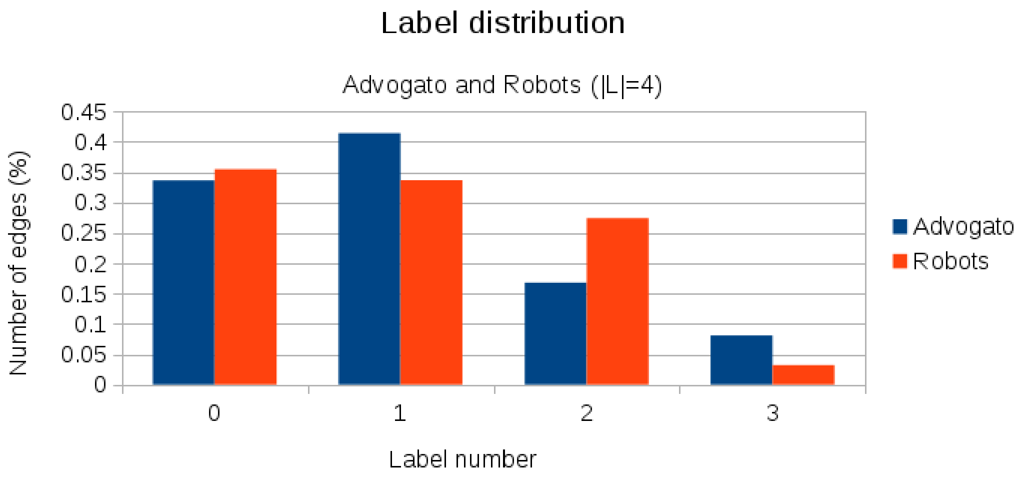


Figure 4.2

- **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. item **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*.
- **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*.

- **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.
- **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. The network contain loops, i.e., self-citations.

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
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
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

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
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
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
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

4.2 Queries

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 synthetic datasets when $8 \leq |sL| \leq 13$ and 2, 4 and 6 otherwise. Each query set consists of 200 or 2000 queries. The first half of these queries should be answered with True and the second half should be answered with False.

For each query set we took into account the difficulty of its queries. The difficulty of a query $q = (s, t, L)$ can be defined in the following way: the number of vertices visited during a labelled BFS from s to t before reaching a conclusion. This difficulty can vary along the minimal distance between s and t in the graph and the number of labels in L and be different for true and false queries. False queries with a large label set may explore much larger sections of the graph than true queries with the same label set. 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.

4.2.1 Query generation

Queries are generated in the following way. A random vertex $v \in V$ and a random minimal difficulty m are chosen. 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 $k = nq/100$, where nq is the desired number of True- or False-queries, random label sets L. Then it is tested whether (v, w, L) is True or False using BFS and how many vertices l where visited during BFS . If $l \geq m$, 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 m can respectively increase and decrease if it takes many rounds to generate the 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.

Chapter 5

Methods

5.1 Taxonomy of all methods

In this chapter we give a taxonomy of all the methods that we have used or strongly considered during our experiments for solving LCR-queries. We give a pseudocode and describe modifications or variations of the original idea.

5.1.1 General comments

Graph $G = (V, E, \mathcal{L})$ is a labelled graph. An edge from v to w can be written as (v, w) or (v, w, l) where label $l \in \mathcal{L}$.

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.

Let Ind be an index for all nodes $v \in V$, let $Ind(v)$ be the index for node v and let $Ind(v, w)$ be a list of (minimal) label sets connecting v and w. The index could be a full (or exact) or partial index. A full index can answer any query (s, t, L) can solely by using the index. A partial index might give a false positive or negative and possibly needs some exploration of the graph.

If $(w, L) \in Ind(v)$ and Ind is a full index, this means that query (v, w, L) is true, as well as any query (v, w, L') with $L \subseteq L'$. Note that L and L' are label sets.

$Ind(v)$ is said to be minimal if for any two different pairs (u, L) and (u, L') in $Ind(v)$ 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 $Ind(v)$ is minimal.

An index can have two modes: blocked mode and non-blocked mode. In blocked mode we store the index as a triple-array with dimensions: $|V| \times |V| \times |\mathcal{L}|$. 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.

5.1.2 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 $O(|V| + |E|)$. The variable marked at line **1** has been implemented using a bitset,

Algorithm 2 BFS(s, t, L)

```
Let $marked : V \longrightarrow \{0,1\}$ be a mapping
Let $q$ be a queue
$q$.push($s$)
while $q$ is not empty do
    $v \leftarrow q$.pop()
    $marked[v] \leftarrow 1$
    if $v = t$ then
        return True
    end if
    for $(v, w, l) \in E \wedge marked[w] = 0$ do
        if $l \in L$ then
            $q$.push($w$)
        end if
    end for
end while
return False
```

which makes it possible to modify and check for any value in constant time rather than in $O(\log(N))$.

5.1.3 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. LandmarkedIndex builds a complete index for a subset $V' \subseteq V$ of $|V| = k$ nodes, also called landmarks.

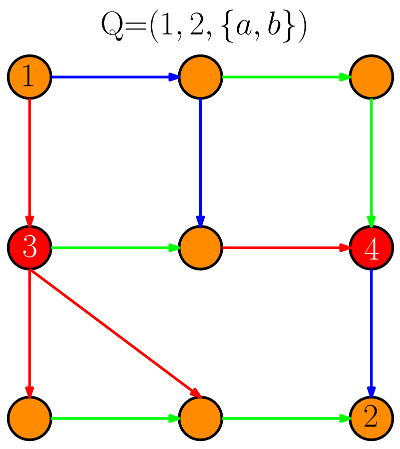


Figure 5.1: The query is $(1, 2, \{a, b\})$. The red edges are a, the blue edges b and the green edges c. Also let the red vertices 3 and 4 be landmarks. The query resolves to true as 1 can reach 4 and 4 can reach 2.

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 LandmarkedIndex as the Landmarked-index without extensions, LandmarkedIndex1 with only the first extension, LandmarkedIndex2 with only the second extension and LandmarkedIndex12 with the first and second extension. We refer to LandmarkedIndex with only the third extension as LandmarkedIndex3 and with both the first and the third extension as LandmarkedIndex13 . The third extension uses the same principle as the first extension and hence is a substitute for the first extension.

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 $\langle v_0, v_1, \ldots, v_n \rangle$ 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 isL that maps any vertex $v \in V$ to either -1 if it is not a landmark and a positive integer in the range $0 \ldots k$ if it is 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 $Ind(v_i)$ and returns True if and only if this succeeded. TRYINSERT only succeeds whenever L is incomparable to any other L' in $Ind(v_i, u)$ or whenever L is only a subset of some of the entries in L'. Moreover, it removes any superset L' of L in $Ind(v_i, u)$. Hence, TRYINSERT preserves the minimality of $Ind(v_i, u)$. 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:   $(u, L) \leftarrow q$.pop()
5:   if tryInsert($u, v, L$) = False then
6:     continue
7:   end if
8:   if hasBeenIndexed($u$) = True then
9:     forwardProp($v, u, L$)
10:    continue
11:  end if
12:  for $(u, w, l) \in E$ do
13:    if $l \in L$ then
14:      $q$.push($w, L \cup \{l\}$)
15:    end if
16:  end for
17:  $hasBeenIndexed(v) \leftarrow$ True
18: end while
```

Algorithm 4 tryInsert(u, v, L)

```
1: if $v = u$ then
2:   return True
3: end if
4: if $\exists[(u, L') \in Ind(v) | L' \subseteq L]$ then
5:   return False
6: end if
7: remove any $(u, L')$ s.t. $L \subset L'$ from $Ind(v)$
8: add $(u, L)$ to $Ind(v)$
9: return True
```

Algorithm 5 forwardProp(v, u, L)

```
1: for $(w, L') \in Ind(u)$ do
2:     tryInsert($v, w, L \cup L'$)
3: end for
```

Propagation

There are a few variations/extensions of the basic idea. Two of these have to do with 'propagating' the index. This more or less revolves using the index of a node v_i for an ascendant node w of v_i. When there is a path P from w to v_i, we can say that any entry of v_i can be copied into w appending $Labels(P)$. A similar thing was as optimization in Zou's algorithm [10].

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.

In our code we used by default 'forward propagation'. While building the index for a node v_i, we might visit a node u which has already been full indexed. We can try to insert all u's entries to v_i appending L to these entries. Also we do not have to add any out-neighbours of u to q, as these have been already indexed.

During construction we maintain a mapping that indicates which vertices $v \in V$ have already been (fully) indexed by using $hasBeenIndexed$.

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 DoubleBFS 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 $Ind(v_i)$ 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 default.

First extension

The basic idea for the first extension is the following. Given a query $q = (v, w, L)$ we might stumble upon a landmark v' either by exploring a part of the graph or using an entry obtained by the first extension. If the 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.

In figure 5.1 we see that 1 can reach landmark 3, but 3 cannot reach 2 over $\{a, b\}$. However, 3 can reach two other vertices. If one of these vertices were able to reach 2 then 3 could reach 2. Hence, there is no point in evaluating these vertices during a BFS and these 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 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 call of Algorithm 3. As long as $|L| = 1$ and $u \neq w$ we insert u to $LP[w][l]$ 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.

Second extension

The second idea creates a number of entries (a pair of a label set and a destination landmark) for all non-landmark vertices. This is done after all landmark vertices have been fully indexed. Each non-landmark node $v \in V$ has a budget b and indexes up to b minimal paths to landmarks v'. After it has found this many paths or can't find any more paths, it stops. The algorithm for this part is very similar to Algorithm 3. The differences are that we only add entries (u, L) to $Ind(v)$ if u is a landmark, that we stop after having successfully inserted b entries to $Ind(v)$, that 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, and that we traverse each $u \in V$ at most once.

Third extension

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.

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 $e \in seqE$ consists of a bitset b and a label set L. Each bitset is an array 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.

The third extension tweaks Algorithm 3 a little. Each time an entry (u, L) has been discovered for $Ind(v)$ we create a new entry e for L and set the bit belonging to u in the bitset of e to 1 or only do the latter in case there is already an entry for L. At the end we merge the entries, i.e. given two entries e with label set L and e' with $L \cup \{l\}$ we join the bitset of e with e' as e' can reach all entries of e.

The query algorithm is very similar to the first extension. Rather than doing a single extensive query for the first landmark found, we run one extensive query for each landmark found.

The advantage of the first extension 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 because of the fact that large label sets can only be used a small minority of the queries. Another advantage is that we can quickly join the bitset used by the query-algorithm with the entries in $seqE$. Joining can be done in $\frac{N}{64}$ time on a 64-bit machine.

Query algorithm

Algorithm 6 is the pseudocode for resolving a query for LANDMARKEDINDEX2 . First we try to resolve the query directly if v is a landmark. Otherwise we try to look for all landmarks v' that we can reach from v using v's b entries. For any such v' we try to resolve the query directly. If all of this fails as well, we continue to traverse the graph using a BFS. Only when we hit a landmark during this BFS we try to resolve the query directly. When a query fails we do not need to look to any of the out-edges of u as w cannot be reached from u using label set L in this case.

The difference between LANDMARKEDINDEX1 and LANDMARKEDINDEX12 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 LANDMARKEDINDEX1 and LANDMARKEDINDEX13 is that the first call of QUERYDIRECT of any landmark has been replaced by 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.

LANDMARKEDINDEX12 : Running time and memory

The analysis is for LANDMARKEDINDEX12 with k landmarks and b budget per non-landmark node.

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

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'.

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

The running time of the index construction algorithm considering only the landmarks is at most $O(|V| \cdot k \cdot 2^{2 \cdot |\mathcal{L}|})$. Each landmark 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 . 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 $Ind(u)$ 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

Algorithm 6 queryLM(v, w, L)

```
if $isL(v) \geq 0$ then
    return queryDirect($v, w, L$)
else
    Let $marked : V \rightarrow \{0, 1\}$ be a mapping
    for $v' \in \{v' | (v', L) \in Ind(v)\}$ do
        // $v'$ is always a landmark
        for $L' \in Ind(v, v') \wedge L' \subseteq L$ do
            if queryDirect($v', w, L$) = True then
                return True
            end if
            $marked[v'] \leftarrow 1$
        end for
    end for
    Let $q$ be a queue
    $q$.push($v$)
    while $q$ is not empty do
        $u \leftarrow q$.pop()
        $marked[u] \leftarrow 1$
        if $u = w$ then
            return True
        end if
        if $isL(u) \geq 0$ then
            if queryDirect($u, w, L$) = True then
                return True
            end if
            continue
        end if
        for $(u, u', l) \in E \wedge marked[u'] = 0$ do
            if $l \in L$ then
                $q$.push($u'$)
            end if
        end for
    end while
    return False
end if
```

Algorithm 7 queryDirect(v, w, L)

```
// $v$ is a landmark
for $L' \in Ind(v, w)$ do
    if $L' \subseteq L$ then
        return True
    end if
end for
return False
```

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.

For the non-landmarked nodes, we can make at most b calls to TRYINSERT while traversing the graph. 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 $O(b \cdot 2^{|\mathcal{L}|} + |V|)$.

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 $(u, \{l\}$

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

```
// first extension
if queryDirect($v, w, L$) = True then
    return True
end if
for $l \in L$ do
    for $v' \in LP[v][l]$ do
        $marked[v'] \leftarrow 1$
    end for
end for
// third extension
if queryDirect($v, w, L$) = True then
    return True
end if
for $(b, L') \in seqE$ do
    if $L' \subseteq L$ then
        $marked = marked \cup b$
        break
    end if
end for
```

to $LP[v'][l]$ takes $O(\log(|V|))$. The number of times this happens cannot be more than the number of calls to tryInsert on line **5**.

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

The running time of Algorithm 6, i.e. 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 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**.

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 $\Leftrightarrow \exists[(w, L') \in Ind(v)|L' subseteq L]$.

Theorem 5.1.1. *Let $v, w \in V$ and let $L \subseteq \mathcal{L}$. Let $Ind(v)$ be the index created by Algorithm 3 for a given $v \in V$. We have that **(1)** $query(v, w, L)$ = True $\Leftrightarrow \exists[(w, L') \in Ind(v)|L' \subseteq L]$ and that **(2)** $query(v, w, L)$ = False $\Leftrightarrow \neg\exists[(w, L') \in Ind(v)|L' \subseteq L]$.*

Proof. Assume that $query(v, w, L)$ = True. Then there is a L-path P from v to w. We must establish that $\exists[(w, L') \in Ind(v)|L' \subseteq L]$. We do this by induction on the set of landmarks V'

Base case: $hasBeenIndexed(u)$ is false for all $u \in V'$.

An entry (u, L') is only pushed on q if v has a L-path to u. Hence the only reason for $\neg\exists[(w, L') \in Ind(v)|L' \subseteq L]$ is that the if-statement on line **5** or **8** was true at some point not allowing such entry to be added.

In the first case, let x be the vertex and L' the label set s.t. tryInsert(v, x, L') is false. This implies that there exists (x, L^*) in $Ind(x)$ and that $L^* \subseteq L'$. Hence in this case $\exists[(w, L') \in Ind(x)|L' \subseteq L]$.

The second case does not apply in the base case.

Step: $hasBeenIndexed(u)$ is true for some $u \in U \subseteq V'$. We assume that for any $u \in U$ if $query(u, w, L)$ True then $\exists[(w, L') \in Ind(u)|L' \subseteq L]$.

In the second case, let u be the vertex s.t $hasBeenIndexed(u)$ is true. We can assume that for any $u' \in V$ and $L_1 \subseteq \mathcal{L}$ that $query(u, u', L_1)$ = True $\Leftrightarrow \exists[(u', L_2) \in Ind(u)|L_2 \subseteq L_1]$.

Assume that $\exists[(w, L') \in Ind(v) | L' \subseteq L]$. We must establish that $query(v, w, L) = \text{True}$. An entry (u, L') is only pushed on q is v can reach u using label set L'. The presence of the entry hence implies that there is a L'-path from v to u. Hence we have that $query(v, w, L) = \text{True}$.

The fact that **(1)** is true implies that **(2)** is true. □

DoubleBFS

DoubleBFS is a special case of LandmarkedIndex , i.e. $k = N$. Any extension is irrelevant in this case and any query is answered by queryDirect .

5.1.4 Partial

PartialIndex also builds an index for a subset of k nodes like LandmarkedIndex , 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 LandmarkedIndex 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 $O(\sqrt{(N)})$.

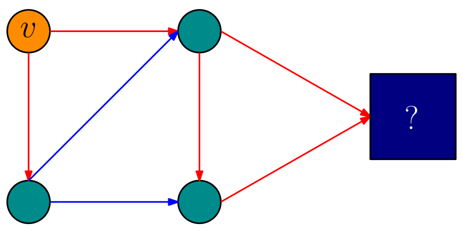


Figure 5.2: Suppose we start a query at orange vertex v. Let the box on the right be a large graph and the cyan-colored vertices be vertices for which there is a partial index. Each of these also has a parital index. Having to run a direct query for each of these will most likely yield the same result.

5.1.5 NeighbourExchange

This method is based on Algorithm **??** in Chapter 2 with some differences applied. The algorithm builds a full index. Algorithm 9 shows a pseudocode. 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 an Upd and an 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.

5.1.6 Joindex

This method is based on the discussion in Section 3.7 with some differences applied. Joindex can both be used with intersection or union.

Algorithm 9 NeighbourExchange()

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

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. $u \in Out(v) \cap In(w)$. 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. 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.

2. Joindex only decreases the index size but not the construction time. LandmarkedIndex 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 $O(|V| + |E|)$.

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

```
if $|L| = 0$ then
  return False
end if
if $v = w$ then
  return True
end if
for $u \in (Out(v) \cap In(w))$ do
  if $\exists[(u, L') \in Out(v) | L' \subseteq L]$ then
    if $\exists[(u, L') \in In(w) | L' \subseteq L]$ then
      return True
    else
      if $\exists[(u, L') \in In(w) | L' \supseteq L \wedge J(u, L')]$ then
        marked[$u$] $\leftarrow 1$
        JoindexQueryUnion($u, w, L, marked$)
        marked[$u$] $\leftarrow 0$
      end if
    end if
  else
    if $\exists[(u, L') \in Out(v) | L' \supseteq L \wedge J(u, L')]$ then
      marked[$u$] $\leftarrow 1$
      JoindexQueryUnion($v, u, L, marked$)
      marked[$u$] $\leftarrow 0$
    end if
  end if
end for
return False
```

5.1.7 ClusteredExact

This method is based on the Section in Chapter 2 where we observed that separating the graph into some subgraphs and building a full index for these subgraphs is more efficient.

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. cID 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 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.

5.1.8 Best effort Zou

This method is a best effort approach to implement Zou's algorithm [10]. 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.
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.
3. We find the in- and out-portals of each C_i, i.e. we find those vertices $v \in C_i$ s.t. there exists $(w, v) \in E$ or $(v, w) \in E$ where $w \notin C_i$. For each in- or out-portal we generate a replica node v'. Between the two there is always 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.
4. 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 l are found using M_{C_i}. Each (v, w, l) is added to a graph D.

Algorithm 11 ClusteredExactQuery(v, w, L)

```
1: Let $marked : V \to \{0, 1\}$
2: Let $q$ be a queue
3: while $q$ is not empty do
4:   $x \leftarrow q$.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:  $marked[x] \leftarrow 1$
14:  for $p \in outP[cID[x]]$ do
15:    if directQuery($x$,$p$) = True then
16:      for $(p, s, l) \in E \wedge l \in L$ do
17:        push $s$ onto $q$
18:      end for
19:    end if
20:  end for
21: end while
```

5. We topologically sort D and reverse the order.

6. The index is propagated upwards according to the topological order for all vertices in D. Hence, $M_G(u, -) = Prune(\bigcup_{(u,v)\in E_D}(M_{C_i}(v, -)))$.

7. For each out-portal u_i that is part of SCC_i we look for all inner vertices u_j of any SCC_j with $j \neq i$ and set $M_G(u_i, -) = Prune(M_G(u_i, -) \odot M_G(u_j, -))$. 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.

8. For each inner vertex u_i in SCC_i we look at all out-portals p_i and set $M_G(u_i, -)$ to $Prune(M_G(u_i, -) \odot M_G(p_i, -))$. This copies all the entries from the out-portal.

9. For each inner vertex u_j with $j \neq i$ we look for an in-portal p_i and a vertex u_i and set $M_G(u_i, -)$ to $Prune(M_G(u_i, -) \odot (M_G(p_i, -) \odot M_G(u_i, -)))$.

Note that $M_G(u, -)$ is the same as $Ind(u)$ in our implementation. $M_G(u, -) \odot M_G(v, -)$ is implemented by looking at any $(v, L) \in Ind(u)$ and concatenating (v, L) to any $(w, L') \in Ind(v)$ which results in $(w, L \cup L')$. $(w, L \cup L')$ is attempted to be added to $Ind(u)$ by using TRYINSERT .

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.

We decided not to implement the optimization in the end, because the results without the optimization were already quite disappointing, i.e. they were not competitive to e.g. DOUBLEBFS . 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 LANDMARKEDINDEX , i.e. sort the entries of the heap according to the number of labels in the label set.

5.2 Index maintenance

In this section we stress the issue of index maintenance. Suppose that after running LANDMARKEDINDEX , PARTIALINDEX , DOUBLEBFS , CLUSTEREDEXACT or NEIGHBOUREXCHANGE we have built an index $Ind(v)$ 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 3.3 storing some redundant information might help making maintenance more easy.

Figure 5.3 illustrates a problem 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.

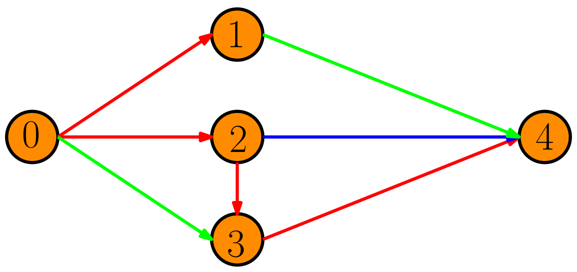


Figure 5.3: A simple graph, where edge color indicate the label. Let red be $\{a\}$, blue be $\{b\}$ and green be $\{c\}$. Vertex 0 can reach vertex 4 over several paths, but only has to store $\{a\}$, because all other paths are supersets of that path.

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

5.2.1 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 $Ind(v)$ or copy entries from a landmark v' that has been fully indexed.

5.2.2 Removing an edge

Removing an edge (v, w, l) consists of two steps. First we need to remove any entry $(w', L) \in Ind(v')$ 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

Algorithm 12 insert(v, w, l)

```
 1: add $(v, w, l)$ to $G$
 2: reset $hasBeenIndexed$
 3: $hasBeenIndexed(w) \leftarrow$ 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:     $x \leftarrow q$.pop()
11:     if $x$ is a landmark then
12:         LabelledBFSPerNode'$(x)$
13:     end if
14:     for $(v, x) \in E$ do
15:         $q$.push$(v)$
16:     end for
17:     $hasBeenIndexed(x) \leftarrow$ True
18: end while
```

entries. A call to LabelledBFSPerNode started at node w assumes that if $(v, L) \in Ind(w)$ we need not to look at any descendants of v, but this might be wrong if $(v, L) \in Ind(w)$ but one of its descendants v' does not have an entry in $Ind(w)$ 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 $w' \in V$. For any $(v', L) \in Ind(w)$. 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 $\{v|(v, L) \in A\}$.

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 $\{(w', L)|w' \in A'_V \wedge (v', w', l') \in E \wedge L \in \{\{l'\} \cup L'|(v', L') \in (Ind(y)\}\}$. In this case the queue consists of all entries $(w', L' \cup \{l\})$ s.t. there is an in-neighbour v' with an entry $(v', L') \in Ind(y)$ This solution would give errors.

5.2.3 Changing edge label

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

5.2.4 Adding a node

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

5.2.5 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.

5.3 Extensions

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

5.3.1 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 $m : |V| \longrightarrow \{0, 1\}$. m has the same role as the variable *marked* in the description of BFS (Section 5.1.2).

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.

LandmarkedIndex 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 $m(v) \leftarrow 1$.

5.3.2 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. $d = min_{P \in P(v,w) \wedge Labels(P) \subseteq L}(\#$
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. LandmarkedIndex 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 $d_1 < d_2 when L_2 \subset L_1$. 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.

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

The goal of any index in the end is to speed up queries compared to BFS . Hence we discuss this for any experiment.

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. 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.

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

6.1.1 Datasets and methods

We used the methods PartialIndex , LandmarkedIndex2 ($k = N/10, b = k/10$), LandmarkedIndex12 ($k = N/10, b = k/10$) 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.

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 LandmarkedIndex12 with $k = N$ (last column). The small difference has to do with the use of the second extension by LandmarkedIndex12 . Especially in the cases of a uniform distribution we see that the construction time of Zou explodes.

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 5.1.8). 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 a slightly more efficient implementation should be possible. However, as the steps before these steps, i.e. steps 1 to 6, already take more time than LandmarkedIndex or PartialIndex

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

, we do not believe it can compete with the other methods. 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 LandmarkedIndex (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 LandmarkedIndex . The results for ClusteredExact are staggering. A 0.01 in the table means that the average query 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.

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

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 LandmarkedIndex12 are slightly above those of LandmarkedIndex2 in some cases and LandmarkedIndex2 is clearly better in the remaining cases. There is not a clear winner here.

The speed-ups achieved by LandmarkedIndex12 with $k = N$ are way higher than all the other speed-ups. The actual query answer times are very low for LandmarkedIndex12 (many in the range of 10^{-7}. There might be some inaccuracy here. LandmarkedIndex12 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.

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.

Table 6.4: Total speed-up per condition (query set and True/False-queries) for LandmarkedIndex2 ($k = N/10$) 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
LandmarkedIndex2	(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
LandmarkedIndex12	(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

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: LandmarkedIndex2 ($k = N/20, b = 20$), LandmarkedIndex2 ($k = N/20, b = 0$), LandmarkedIndex12 ($k = N/20, b = 20$) and LandmarkedIndex12 ($k = N/10, b = 20$). We fixed the budget either to 20 or to 0. In the case $b = 0$, one could consider LandmarkedIndex2 to be the same as LandmarkedIndex .

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 LandmarkedIndex12 ($k = N/10, b = 20$

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. If we compare **advogato** (roughly $5k$ edges) against the PA-datasets with $5k$ edges, then we see that the construction times are comparable to that of **V5kD2L8exp**. However if we do the same for **arXivheppL8exp** (34k edges) and **V25kD2L8exp**, we see that the first dataset needs more time. This is explained by the fact that **arXivheppL8exp** has a more evenly distributed out-degree. This explains why it has a higher construction time in any case (except BFS) than **p2p-GnutellaL8exp** despite the fact that the latter one has more nodes (63k edges). Hence the number of vertices is not the most important factor here.

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.

The FF- and PL-datasets take considerably less time to build than their ER- counterparts with the same label distribution. The difference with PA is smaller. We do wish to note that the PL-datasets have a degree of roughly 3.6. The top section of Table 6.6 shows the real datasets. An interesting observation here is that construction

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

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 3 minutes to build. In contrast, **arXivhepphL8exp** took way more time 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 landmarks have already been constructed and a landmark v needs to be constructed, the chances are higher in a graph like **p2p-GnutellaL8exp** that it will stumble onto one of those K landmarks and can use forward pruning than in a graph like **arXivhepphL8exp** .

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.

When we only look at the datasets with an exponential distribution, PA, PL and FF have a similar growth in index size. However, if we also look at the label distribution, we see quite some differences for the PA-datasets. The ratio of the index sizes between the normally distributed and the exponentially distributed datasets grows marginally from 2.03 ($5k$) to 2.05 ($125k$). For uniformly distributed datasets this ratio grows from 2.40 ($5k$) to 2.42 ($125k$). The index construction times did not exhibit this marginal growth.

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 LandmarkedIndex12 ($k = N/20, b = 20$).

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 LandmarkedIndex12 ($k = N/20, b = 20$).

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' having $25,000$ edges and a degree of either $3,4$ or 5 and at least 8 labels and an exponential label distribution, we only ran LandmarkedIndex12 ($k = N/20, b = 20$). 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] (see Figure 3.5). 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 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, but this was for some specific large datasets with few landmarks. Therefore we will not refrain from using the first extension in future experiments and keep using it.

The difference between LandmarkedIndex2 and LandmarkedIndex2 with $b = 0$ is larger though. One could consider the latter one simply as LandmarkedIndex . Looking at the third and fifth column there is a significant difference in favour of LandmarkedIndex2 ($b = 20$) in all cases. The ratio in favour of LandmarkedIndex2 with $b = 20$ is in most cases around 2. This makes the advantage of the first extension clear. In exchange for a minimal extra amount of construction time and index size we gain quite some improvement. For False-queries the results are less convincing. The ratio in favour of LandmarkedIndex2 with $b = 20$ is in most cases around 1. Though for the larger datasets, i.e. $|V| \geq 25,000$, this ratio is above 1 in all cases.

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

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 distribution and compare these against the speed-ups for their normal or uniform counterparts, we see that the speed-ups in the first case are higher. However, if the degree of the graph increases, the speed-up for exp-datasets decreases. (explanation + ???) Parts of the differences could also be random effects due to the randomness in the query generation.

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 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 can set the number of landmarks to a lower value 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.

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

dataset	L12		L2		L2$b = 0$		L12'	
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
V25kD5L8exp	859.30	1.95	794.35	1.74	429.71	1.72	837.87	2.01
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

In Figure 6.1 we can see the speed-up per query for **p2p-GnutellaL8exp** and the third query set using LandmarkedIndex 12($k = N/10, b = 20$). The total speed-ups (Table 6.12) are 771.22 and 646.22. About 55% of the True-queries are faster than its average. For False-queries this is 93%. 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.

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

dataset	L12		L2		L2$b = 0$		L12'	
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
V25kD5L8exp	529.14	2.71	515.58	2.07	280.85	2.06	593.65	3.17
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

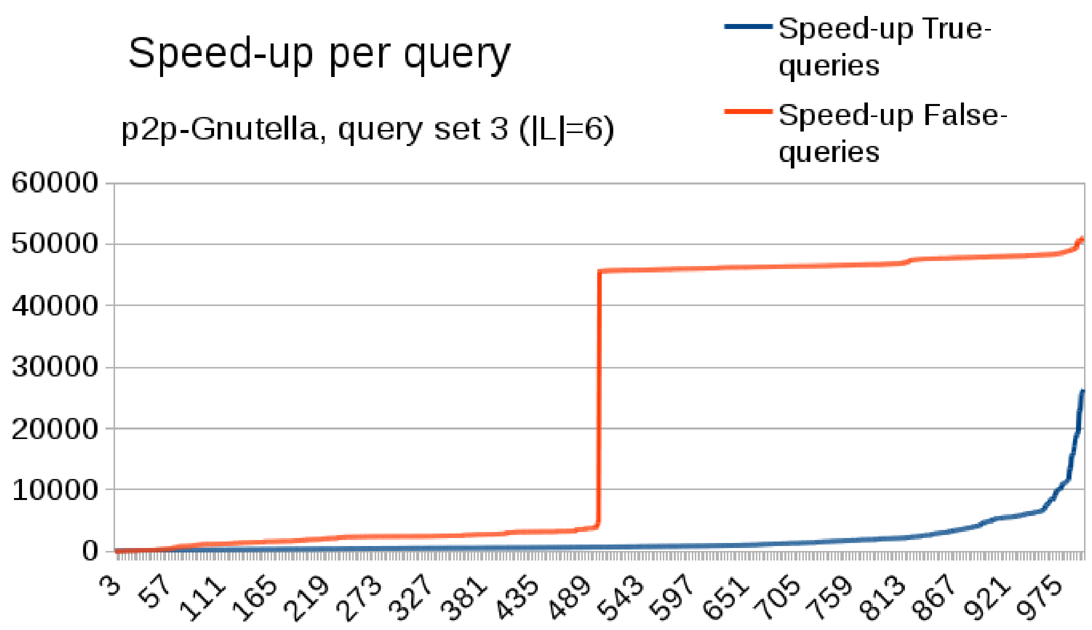


Figure 6.1: Speed-up per query (sorted in ascending order) for **p2p-GnutellaL8exp** and the third query set using LANDMARKEDINDEX 12($k = N/10, b = 20$).

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

dataset	L12		L2		L2$b = 0$		L12'	
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
V25kD5L8exp	516.39	1.92	539.44	1.91	292.29	1.90	550.17	2.88
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 LandmarkedIndex12 ($k = N/20, b = 20$).

degree,$|\mathcal{L}|$	qs_0,true	qs_0,false	qs_1,true	qs_1,false	qs_2,true	qs_2,false
	$|\mathcal{L}|/4$		$|\mathcal{L}|/2$		$|\mathcal{L}| - 2$	
$3, 8$	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 LandmarkedIndex2 with $b = 20$ and LandmarkedIndex12 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 LandmarkedIndex12 with ($b = 20$ and $b = 40$). This is to see whether a larger budget works better or not.

6.3.1 Index construction time and size

In Table 6.15 we see the index construction time (s) and size (MB) for several datasets and LandmarkedIndex12 ($b = 20$) and ($b = 40$). We excluded the data for LandmarkedIndex2 from the top section of the table as it is very similar to LandmarkedIndex12 .

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.

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.

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 whereas **webGoogleL8exp** has a stronger growth in the beginning which grows nearly flat for the last 5% of the landmarks.

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

Dataset	$b = 20$ (s)	$b = 20$ (MB)	$b = 40$ (s)	$b = 40$ (MB)
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

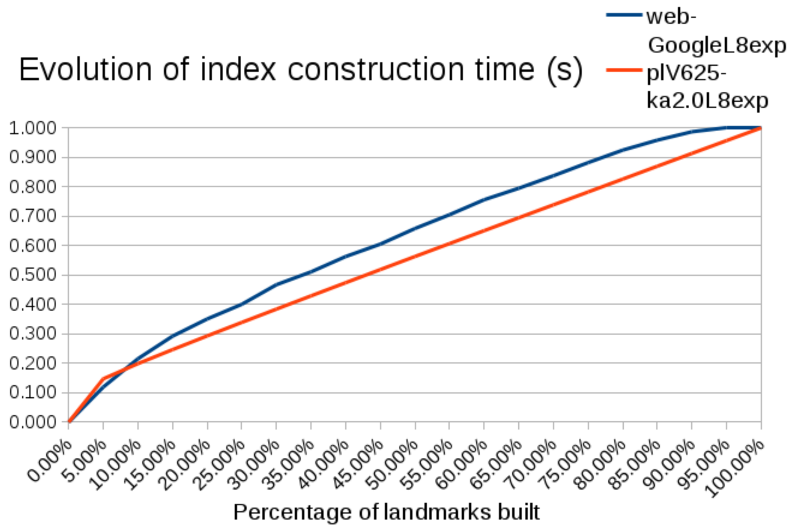


Figure 6.2: On the y-axis we see the percentage of the total time needed to build all landmarks against the percentage of landmarks that have been built on the x-axis. This shows the index growth over time of datasets **plV625ka2.0L8exp** and **webGoogleL8exp** .

6.3.2 Speed-ups

Table 6.17 shows the speed-ups for LandmarkedIndex12 $b = 20$, LandmarkedIndex12 $b = 40$ and LandmarkedIndex2 $b = 20$.

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.

NotreDameL8exp has decent speed-ups in all query conditions. There is a clear difference between the first

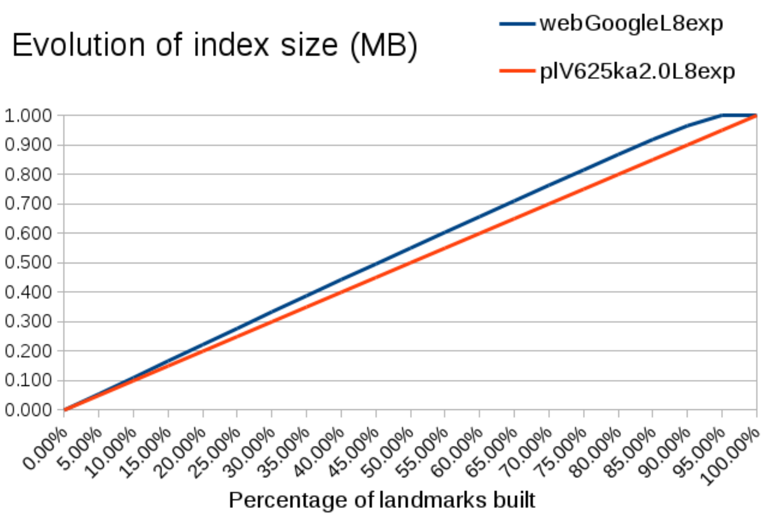


Figure 6.3: On the y-axis we see the percentage of the total size needed to build all landmarks against the percentage of landmarks that have been built on the x-axis. This shows the index growth over time of datasets **plV625ka2.0L8exp** and **webGoogleL8exp** .

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

row (LandmarkedIndex12 $b = 20$) and the third row here (LandmarkedIndex2 $b = 20$). 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 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

Table 6.17: Speed-ups for LandmarkedIndex12 $b = 20$, LandmarkedIndex12 $b = 40$ and LandmarkedIndex2 $b = 20$ for the datasets for which $k = \frac{N}{50}$.

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

Table 6.18: Speed-ups for LandmarkedIndex12 $b = 20$ and LandmarkedIndex12 $b = 40$ for the datasets for which $k = \frac{N}{100}$.

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

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 LandmarkedIndex 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 6.20 shows the difference between "total speed-up" (sum

Table 6.19: Speed-ups for LandmarkedIndex12 $b = 20$ and LandmarkedIndex12 $b = 40$ for the datasets for which $k = \frac{N}{250}$.

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

Table 6.20: For query condition $(qs_0, True)$ and $(qs_0, False)$ and some datasets we show the "total speed-up", "average speed-up" and the percentage of queries for which the individual speed-up is higher than the "total speed-up".

Dataset	$(qs_0, True)$			$(qs_0, False)$		
	total	average	individual%	total	average	individual%
webGoogleL8exp	$6,379.07$	$6,643.92$	0.504	1.43	$14,422.75$	0.287
zhishihudongL8exp	775.97	775.33	0.407	1.00	423.46	0.175
plV625ka2.0L8exp	$14,862.85$	$15,006.39$	0.543	5.09	$8,186.77$	0.763

of all BFS query times over sum of all LandmarkedIndex times), "average speed-up" (average of individual speed-ups) and the percentage of queries for which the "individual speed-up" (BFS against LandmarkedIndex on a single query) is higher than the total speed-up. The results show that even with a low "total speed-up" we can achieve a much higher speed-up for a large part of the queries. For **webGoogleL8exp** we can have that despite a total speed-up of 1.43 we see that 28.7% of the queries has a higher individual speed-up. We know from the raw data that around 28% has an individual speed-up above 10.0.

6.4 Maintenance

We tested the correctness and speed of our maintenance methods by doing the following. First we built a LandmarkedIndex 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 LandmarkedIndex for G lI'. We compared lI and lI' over $L = 200$ random queries. If all these queries were answered correctly, we assume that the method is correct.

6.4.1 Adding an edge

Table 6.21 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, as it this will take around 5% for datasets with a uniform or normal label set distribution and around 22% for datasets with an exponential distribution.

Table 6.21: Percentage of the initial construction time (s) that adding a random edge took for 4 datasets. 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.

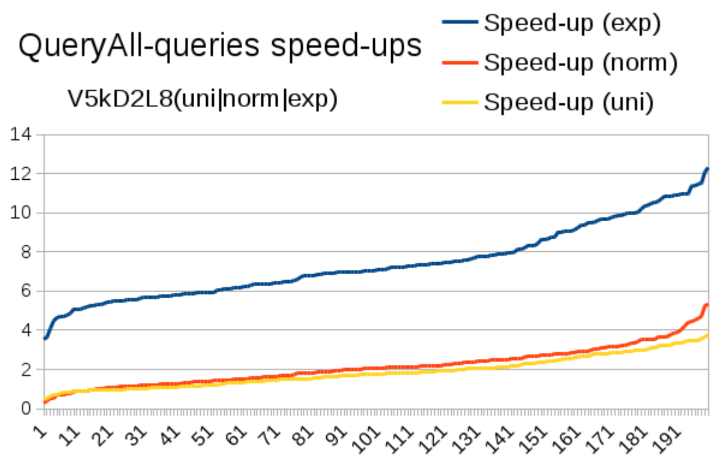


Figure 6.4: Speed-ups per query for QueryAll-queries for PA-datasets with 5, 000 vertices, a degree of 2, 8 labels and either a uniform, normal or exponential label set distribution. We used LandmarkedIndex12 ($k = N/10, b = 0$).

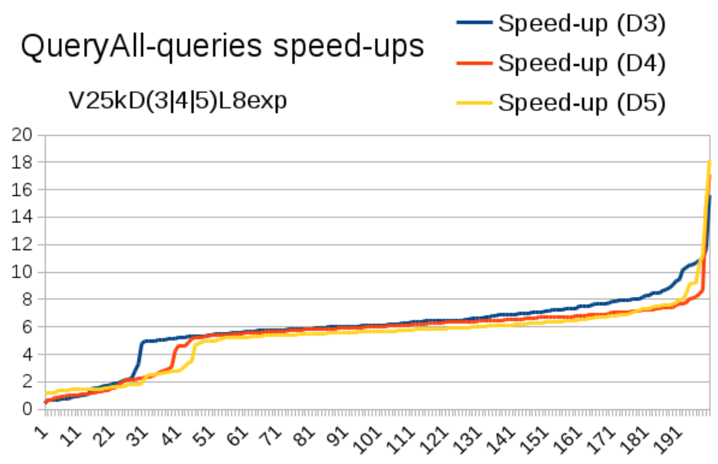


Figure 6.5: Speed-ups per query for QueryAll-queries for PA-datasets with 25, 000 vertices, a degree of 3, 4 or 5, 8 labels and an exponential label set distribution. We used LandmarkedIndex12 ($k = N/10, b = 0$).

Table 6.22: 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

6.5.2 Distance queries

Table 6.23 shows the index size and construction time for two versions of LandmarkedIndex12 ($k = N/10, b = 0$) 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

Table 6.23: Index size (MB) and index construction time (s) for LandmarkedIndex12 and LandmarkedIndex12 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

index size is always at least 2 times larger. The index construction time can even grow much stronger. We do wish to note that this solution returns the exact distance, i.e. no approximation of the distance.

Chapter 7

Conclusion

Bibliography

[1] Francesco Bonchi, Aristides Gionis, Francesco Gullo, and Antti Ukkonen. Distance oracles in edge-labeled graphs. In *a*, EDBT, 2014. 1, 7, 9, 36, 43

[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. 1

[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. 5, 6

[4] George Fletcher and Yuichi Yoshida. Notes on landmark labeling label-constrained reachability queries. 14, 16

[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. 1

[6] Jure Leskovec and Andrej Krevl. SNAP Datasets: Stanford large network dataset collection. `http://snap.stanford.edu/data`, June 2014. 17

[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. 17

[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 &#38; knowledge management*, CIKM '13, pages 1601–1606, New York, NY, USA, 2013. ACM. 1

[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. 1, 5, 6, 7

[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, 9, 11, 13, 14, 26, 32