Karlsruhe Institute of Technology

Engineering an Efficient Reachability Algorithm for Directed Graphs

Diploma Thesis of

Florian Merz

At the Department of Informatics
Institute of Theoretical Informatics
Algorithmics II

Reviewer: Prof. Dr. Peter Sanders

9th December 2013

I declare that I have developed and written the enclosed thesis completely by myself, and have not used sources or means without declaration in the text.

PLACE, DATE

. . . .

(**YOUR NAME**)

Contents

1 Introduction **7**
1.1 Related Work 7
1.1.1 GRAIL 8
1.1.2 Topological Folding 9
1.2 Overview 9

2 Preliminaries **10**
2.1 Graph Definitions 10
2.2 DAG-Properties 11
2.3 Algorithms and Data Structures 12
2.3.1 First-In-First-Out Queue (FIFO) 12
2.3.2 Priority Queue 12
2.3.3 Depth-First-Search 12
2.3.4 Breadth-First-Search 13
2.3.5 Bidirectional Search 13

3 Prune to Reach **14**
3.1 Search Spaces 14
3.2 Topological Levels 16
3.3 DFS-Trees 18
3.4 Base Reachability Query 20
3.5 More Pruning 22
3.5.1 Reverse Topological Levels 22
3.5.2 Peek Nodes 23

4 Construction and Query **27**
4.1 Construction 27
4.1.1 Construction of Search Spaces 27
4.1.2 Setting Topological Levels 29
4.1.3 Construction of DFS-Trees 29
4.2 Query 33

5 Experiments **36**
5.1 Test Data 36
5.2 Experiments on P2REACH 38
5.2.1 Search Spaces 38
5.2.2 Priority Function for Search Spaces 38
5.2.3 Topological Levels and DFS-Trees 40
5.2.4 Query Time Distribution 40
5.3 Comparison with TF and GRAIL 45
5.3.1 Varying Degree and Size 46
5.3.2 Small Real Datasets 50

5.3.3 Large Synthetic Datasets 52
5.3.4 Large Real and Stanford Datasets 53

6 Conclusion and Future Work **55**
6.1 Conclusion 55
6.2 Future Work 55

Bibliography **57**

1. Introduction

A *reachability query* on a directed graph G asks if there exists a path from a node s to a node t. Answering such queries on large graph like datasets has become an issue in various fields of research and real world applications over the past 20 years. Therefore, answering reachbility queries fast and efficiently has become more and more relevant. Nowadays, most XML [2] files make extensive use of ID [11] and IDREF, which transforms their tree based layout into a more complex directed graph. Therefore, querying for reachability requires a reachability query on a directed graph instead of a simple ancestor query. Similarly, the RDF model [20] relies on directed graphs. Various queries on RDF graphs involve reachability, for example to infer the relationship between objects. Since the semantic web builds on RDF, this topic gains more attention as the semantic web becomes more popular.

Network biology uses reachability queries to query for protein-protein interaction on databases like the DIP [33]. Furthermore, reachability plays a role in querying for metabolic pathways on metabolic networks [19] or interaction on gene regulatory networks [3]. Additionally, in the field of model checking [9] reachability queries are needed to check whether a state can reach another state. Similarly, source code analysis uses reachability queries for pointer and dataflow analysis [26, 25].

Since the answer of a reachability query on a directed graph, which contains a cycle covering all nodes is always true, we can reduce a directed graph to its condensation. This is done by calculating the strongly connected components and contracting them into a single node. The condensation is a directed acyclic graph (DAG) which is much smaller in most cases.

The two naive algorithms to answer reachability queries for a DAG are either traversing the graph using a Depth-First Search (DFS) or a Breadth-First Search (BFS) or calculating the transitive closure of the DAG. Simply traversing the DAG results in a query time in $O(m)$ whereas storing the transitive closure needs $O(n^2)$ space and has a complexity of $O(nm)$ to compute but can answer a query in $O(1)$.

1.1 Related Work

Throughout the past years, numerous different approaches for graph reachability emerged. They often combine the previously mentioned two methods for reachability querying. Thus trading off query performance for lower index size while keeping precomputation time low is the challenge of combining aspects of indexing via transitive closure and pure search.

The different approaches can be largely classified into three categories as done by [15]. Those are: *transitive closure compression, hop-labeling* and *refined online search*. Next we will present a short description and notable results for each category.

Transitive Closure Compression

Those approaches calculate a compressed reachability set (TC) for each node u based on a compressed transitive closure. Thus, to answer if a node v is reachable from node u can be answered by checking v against $TC(u)$. Some notable methods based on transitive closure compression are chain representation [14, 5], interval representation [24], dual labeling [32], path-tree [16] and PWAH [31]. On graphs of moderate size these methods tend to perform the best, especially the latest version of path-tree. However, the performance comes with the cost of a large index size which today limits those methods to graphs of a size around one million nodes on reasonable hardware. Additionally, the precomputation time is worse in most cases and can even include the computation of the complete transitive closure.

Hop-Labeling

Methods of the second category gather two sets of nodes for each node u, where the first set contains intermediate nodes that can be reached (L_{out}) and the second one L_{in} contains intermediate nodes that can reach u. To answer a reachability query they use the intersection between L_{out} of the start node and L_{in} of the end node. 2-hop labeling by Cohen *et al.* [10] and 3-hop labeling by Jin *et al.* [18] are approaches which use this method. Hop-labeling can result in a smaller index size than methods of the first category, but result in a slower query answering time and the construction complexity is high, in the case of the original 2-hop labeling approach even in $O(n^3|\text{transitive closure}|)$. Several heuristic speed-up methods [8, 7, 29] have been introduced, but scalability remains a problem of these methods.

Refined Online Search

The third category of methods are online searches on the DAG. They use a precomputation phase to store auxiliary data for edges and nodes which enables aggressive pruning during an online search. GRIPP [30] and Label+SSPI [4] use a tree cover to speed up a DFS. GRAIL [34] uses several random DFS traversals to assign interval labels to each node. Since GRAIL is the state-of-the-art algorithm of this methods we explain it further in Section 1.1.1.

The algorithms of this category have in general a fast precomputation time since they don't need an optimization process or the transitive closure. Thus, they have the disadvantage of a longer query answering time which grows with the size of the input DAG.

1.1.1 GRAIL

GRAIL is a scalable reachability indexing scheme by Yildirim *et al.* [34] which provides a good trade-off between query time and construction time. Their approach requires linear time and space for indexing and results in query times that range from constant time to linear time w.r.t. the order and size of the DAG. GRAIL utilizes multiple random traversals where each traversal yields an interval label for each node. An interval $L_u^i := [r_x, r_u]$ consists of the post-order rank[1] r_u of u and r_x, which denotes the lowest rank of any node x reachable from u. A label $L_v := \{L_v^0, \dots, L_v^d\}$ of a node v has dimension d which is the number of intervals that have been calculated. Let u, v be two nodes with labels L_u, L_v. If there exists one interval in the labels such that $L_u^i \not\subseteq L_v^i$, then v is not reachable from u.

[1] equals *finishTime* in Section 2.3.3

On the other hand, if for all i, $L_u^i \subseteq L_v^i$ holds, u may be reachable from v, but does not have to. Therefore, if for all i, $L_u^i \subseteq L_v^i$, but $v \not\to u$ holds, we call the node u a false positive for the node v. Thus GRAIL uses either an exception list for each node for those false positives, or it utilizes a "smart" online DFS that uses the interval label for pruning. Since calculating exceptions lists generates overhead with respect to precomputation time and index size and they don't scale to very large graphs, GRAIL focuses on the DFS approach.

1.1.2 Topological Folding

TF which stands for topological folding is the latest labeling scheme from Cheng *et al.* [6]. They use *topological levels* [2] which are basically a partition of $V(G)$ into independent subsets $L_1(G), \ldots, L_{\ell(G)}(G)$, where $\ell(G)$ denotes the *topological level number* of G and i of L_i denotes the level of L_i. Furthermore, for any two nodes u, v if $(u, v) \in E(G), u \in L_i(G), v \in L_j(G)$ then $j > i$ holds. They introduce topological folding on topological levels, which in each step deletes the odd levels and preserves reachability by inserting according edges. For edges spanning more than one level, dummy nodes are inserted in order to achieve a correct folding. This process folds G in half with each step until only one level exists. This results in a *topological folding* $\mathbb{G} = \{G_1, \ldots, G_{\ell(G)}\}$ where $G_1 = G$. A $G_i \in \mathbb{G}$ is called *folding graph*. Each node v has a topological folding number $tf(v)$, which is the highest level in which the node is still present.

The labeling scheme of TF maintains two labels for each node v, $label_{in}(v)$ and $label_{out}(v)$. The in-label $label_{in}(v)$ is built by adding v and then recursively adding the nodes in $N^-_{G_{tf(u)}}(u)$ [3] of each node u contained in $label_{in}(v)$. The out-label $label_{out}(v)$ is built analogously. Using this labeling scheme a reachability query s, t is answered positively if $label_{out}(s) \cap label_{in}(t) \neq \emptyset$ and negatively otherwise. Furthermore, they describe how to substitute dummy nodes and also how to handle nodes with high degree.

1.2 Overview

This thesis will describe a scalable efficient algorithm for reachability queries on directed graphs. Our algorithm provides faster construction time on large graphs than other methods with competitive query times and scales to graphs orders of magnitude larger than methods of comparable query performance can handle. The thesis is structured as follows: First we provide preliminaries on directed graphs and algorithms used throughout this thesis. Subsequently we introduce *Search Spaces*, a contraction technique inspired by [13], followed by several pruning and shortcutting techniques in Chapter 3. In Chapter 4, we then describe the algorithm used to preprocess a DAG, in order to obtain the auxiliary data used to speed up a bidirectional breadth-first-search. Furthermore, we introduce our query algorithm in detail.

In Chapter 5 we provide experiments to demonstrate the performance gains of our techniques. Furthermore, we show that our approach scales better than existing methods with similar query performance. Subsequently, we compare our algorithm with the two state-of-the-art algorithms on real world data. Finally, we present the conclusions we drew form those experiments and take a look into future possibilities of our approach.

[2] see Section 3.2 on DAGs for a detailed definition
[3] The out-neighbors of u in the folding graph $G_{tf(u)}$, see Section 2.1

2. Preliminaries

This chapter introduces the basic notations, algorithms and data structures used in this thesis. Most of the notation regarding graph theory is chosen according to [1]. For more detailed information on algorithms and data structures [22] is a good start.

2.1 Graph Definitions

We define a *directed graph (digraph)* as a pair G of sets, where $V(G)$ is a set of elements called *nodes* and $E(G) \subseteq (V(G) \times V(G))$ is a set of ordered pairs of nodes called *edges*. $V(G)$ is called *node-set* and $E(G)$ is called *edge-set*. A digraph is often denoted as $G = (V, E)$.

For an edge $e = (u, v)$, u is called *start-node* and v is called *end-node*. Furthermore, for $e = (u, v)$, u and v are called *consecutive* and u and v are *incident* to e. Throughout this thesis we only regard digraphs without self loops, which are edges $e = (v, v), v \in V$. We use $n = |V|$ to denote the number of nodes and $m = |E|$ to denote the number of edges. $\overline{E} := \{(v, u) \mid (u, v) \in E\}$ denotes the set of reverse edges of G and $\overline{G} = (V, \overline{E})$ is called the reverse graph of G.

A digraph H is called *subdigraph* of G if $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$. Furthermore, given a digraph $G = (V, E)$ and $V' \subseteq V$, then V' *induces* the subdigraph $G(V') = (V', E \cap (V' \times V'))$. Similarly for a digraph $G = (V, E)$ an edge set $E' \subseteq E$ induces a subdigraph $G(E')$ of G, with $E(G(E')) = E'$ and $V(G(E')) = \{v, u \in V \mid (v, u) \in E'\}$.

For a node $v \in V(G)$ we define $G - v := (V \setminus \{v\}, \{(u, w) \in E(G) : \{u, w\} \cap \{v\} = \emptyset\})$ which is G without the node v and without the edges incident to v. This process is called *deleting* v from G.

Furthermore, we define

$$N_G^+(v) := \{u \mid (v, u) \in E(G)\}$$
$$N_G^-(v) := \{u \mid (u, v) \in E(G)\}$$
$$N_G(v) := N_G^+ \cup N_G^-$$

and call $N_G^+(v)$, $N_G^-(v)$ and $N_G(v)$ the *out-neighborhood*, *in-neighborhood* and *neighborhood* of v in G. The *degree* of a node v is defined by $d_G(v) = |N_G(v)|$ and accordingly $d_G^+(v) = |N_G^+(v)|$ and $d_G^-(v) = |N_G^-(v)|$ denote the *out-degree* and *in-degree* of v. For a node $v \in V(G)$ if $d_G^+ = 0$ we call v a *sink*, if $d_G^-(v) = 0$ we call v a *source*. Further we denote the set of all sources of G by $S_{source}(G)$ and the set of all sinks by $S_{sink}(G)$.

We define a *walk* $W := (v_1, v_2, \dots, v_{k-1}, v_k)$ as a sequence of nodes v_i, such that $(v_{i-1}, v_i) \in E(V)$ for $1 < i \leq k$. Since we only consider graphs with unique edges, the W defines a set of edges between the nodes. A walk is called *path* if the nodes $v_1, v_2, \dots, v_k$ are distinct. The length of a walk is the number of its implied edges and a (u, v)-*walk (path)* is a walk (path) between u and v. For a path $C = (v_1, v_2, \dots, v_k)$, $k \geq 3$ and $v_1 = v_k$, we call C a *cycle*. A *directed acyclic graph (DAG)* is a digraph which contains no cycle.

Given a DAG $G = (V, E)$ and two nodes (s, t), we say t is *reachable* from s if there exist an (s, t)-path in G. We denote this by $s \to t$ and if there exists no (s, t)-path in G we denote it by $s \not\to t$.

A *directed tree* is a DAG T with one *root* $r \in V(T)$ and for every node $v \neq r \in V(T)$ exists exactly one (r, v)-path in T. For a tree $T = (V, E)$ we call $\overline{T} = (V, \overline{E})$ a *reverse directed tree*.

Given a digraph G and an edge $e = (u, v) \in E(G)$. The *contraction* of $e = (u, v)$ results in a digraph G' with $V(G') = \{v_e\} \cup V(G) \setminus \{u, v\}$ and

$$\begin{aligned} E(G') = &\{(x, y) \in E(G) \mid \{x, y\} \cap \{u, v\} = \emptyset\} \\ &\cup \{(v_e, x) \mid (v, x) \in E(G) \text{or} (u, x) \in E(G)\} \\ &\cup \{(x, v_e) \mid (x, v) \in E(G) \text{or} (x, u) \in E(G)\}. \end{aligned}$$

We call a digraph G *strongly connected* if for every pair u, v of nodes an (u, v)-walk and a (v, u)-walk exist. A maximal induced subdigraph which is strongly connected is called *strong connected component*. If we contract every strong connected component into a single vertex, we obtain a DAG which is called *condensation* or *SCC-graph* of G.

Throughout this thesis we consider a graph to be finite.

2.2 DAG-Properties

A DAG G has at least one sink and one source. To prove this, take any node $v \in V(G)$, then either v is already a sink or it has an outgoing edge (v, w), following this edge we can recursive apply this statement. Since a DAG contains no cycle and we only consider finite graphs the recursion has to stop, hence we find a sink. The same argument applies to the existence of a source analogously.

A DAG has a *topological sorting*, which means that a bijective function $f : V(G) \to \{1, \dots, n\}$ exists, such that for each edge $(u, v) \in E(G) : f(u) < f(v)$ holds. A simple algorithm to obtain a topological sorting and proof its existence would be the following:

```
1 begin
2   V' ← V
3   order ← 1
4   for i from 1 to n do
5     v ← GetASource(V')
6     SetOrder(v, order)
7     V' ← V' \ {v}
8     order ← order + 1
```

Algorithm 2.1: GetTopologicalSorting

As this algorithm successively deletes sources, we know for any edge $(u, v) \in E(G)$ that u has to be deleted before v can be deleted. Therefore u will receive a lesser order than v.

An rather simple but in our case important property of a DAG G is that for any node $v \in V(G)$, $V' = V \setminus \{v\}$ and $E' = E \setminus \{(u, w) \in E \mid u = v \vee w = v\}$, $G' = (V', E')$ is still a DAG. In words deleting a node $v \in V(G)$ from a DAG G results in a DAG.

2.3 Algorithms and Data Structures

In this section we describe the algorithms and data structures we use throughout this thesis.

2.3.1 First-In-First-Out Queue (FIFO)

A *FIFO* is a data structure that manages a set of elements. It allows to access elements in the same order they where enqueued. In general it provides four simple operations:

push_back Inserts an element at the end of the FIFO

pop_front Removes the first (and therefore oldest) element of the FIFO.

first Returns the first element of the FIFO.

size Returns the number of elements, that are currently contained in the FIFO.

A simple *pop* operation often combines pop_front and first in one step.

2.3.2 Priority Queue

A *Priority Queue* is similar to a FIFO queue but the elements of a priority queue are associated with a *priority*. Instead of the oldest element it returns the element associated with the highest priority that currently resides in the queue. A priority queue provides the following operations:

push Insert an element with an associated priority into the queue.

deleteMin Return the element associated with the highest priority and remove it from the queue.

Furthermore, most priority queues provide operations to initialize a priority queue on a given set of elements and priorities, which can be more efficient than just inserting the elements consecutively. A *peek* operation is common as well. The peek operation returns the element associated with the highest priority without deleting it.

There exist many data structures implementing priority queues, like binary heaps and Fibonacci heaps [12]. See [27] for a comparison. We decided to use *sequence heaps* introduced by [28] in our implementation, as they are cache efficient and perform well on large inputs.

2.3.3 Depth-First-Search

A *Depth-first-search (DFS)* on a directed graph G is a way to traverse a subgraph of G starting at a node $s \in V(G)$. Starting at node s we traverse the fist outgoing edge $e = (s, v) \in E(G)$. We mark the start-node of e and then continue at v. We don't traverse to marked nodes and we return to the previous node only when no unmarked out-neighbors are left for the current node. During the traversal we can number the nodes in the order they where visited, this number is called DFS number, denoted by *dfsNum*. Furthermore, we can number the nodes in the order they were finished (last visited, before returning to their predecessor). This number is denoted by *finishTime*. Algorithm 2.2 implements a simple DFS, which is initially started on a start node v, and an initial *dfsPos* and

finishTimer of 0. We can traverse G completely be starting a DFS for each source of G keeping the marks. See Section 3.3 and [22] for further details.

```
Data: DAG G, dfsPos, finishTimer
1 begin
2     mark v
3     dfsNum(v) ← dfsPos
4     dfsPos ← dfsPos + 1
5     for (v, w) ∈ E(G) do
6         if w is not marked then
7             DFS (w)
8     finishTime(v) ← finishTimer
9     finishTimer ← finishTimer + 1 return dfsPos
```

Algorithm 2.2: DFS(v)

2.3.4 Breadth-First-Search

Breadth-first-search (BFS) is another method to traverse a subtree G' of a digraph G. It traverses G' *layer* by layer, where a starting node s defines layer 0. All nodes in $N^+(s)$ form layer 1. Furthermore, the out-neighborhoods of the nodes in layer i, that have not been traversed yet, form layer $i+1$. In order to keep track of the nodes in the next layer BFS needs a queue to maintain the nodes of the current layer and the next layer.

2.3.5 Bidirectional Search

Bidirectional Search is a speedup technique generally used in the field of shortest path queries. To answer a query for a path between two nodes s and t we search from s forward and from t backward until both searches have hit a common node. In most cases this method reduces the total search space and is often combined with other speedup techniques.

3. Prune to Reach

In this chapter we present our approach Prune to Reach (*P2REACH*). We use a bidirectional BFS on a divided forward and backward search space. During the traversal we prune the search spaces using precalculated auxiliary node data, which includes topological orders and precalculated DFS-Trees. First we explain search spaces in Section 3.1, next the topological levels in Section 3.2 followed by shortcuts and pruning via DFS-Trees Section 3.3 and finally we present a base query algorithm in Section 3.4. Furthermore, in Section 3.5 we explain additional methods for pruning using the previous introduce auxiliary data.

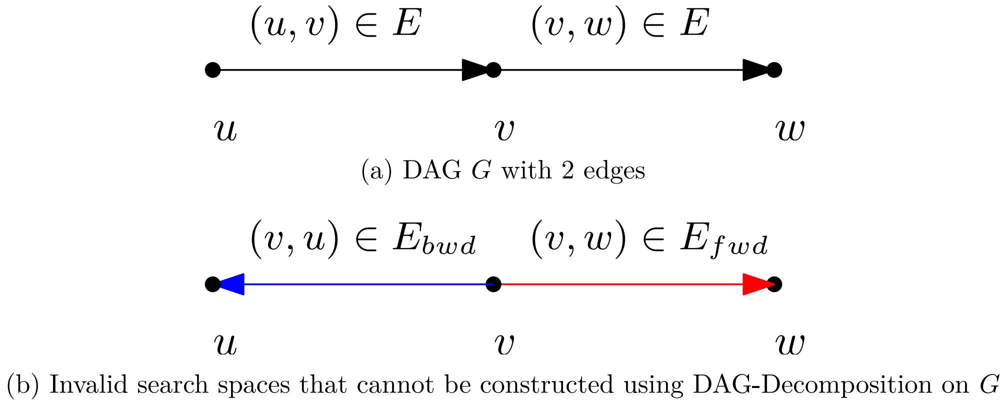
3.1 Search Spaces

As we use a bidirectional BFS we will show that we can restrict the forward and backward searches to their own edge disjoint search spaces. The motivation behind this is to lower the branching factor of a search space during traversal. Therefore, we introduce a forward search space E_{fwd} and a backward search space E_{bwd}. Each edge $e \in E(G)$ is either in E_{fwd} or its reverse edge $\overline{e}$ is in E_{bwd}. While assigning the edges to E_{fwd} and E_{bwd} we have to assure that the forward and backward search have at least one common node for each positive query, which can even be the source or the target. Next we present the assigning scheme and then prove that a simple bidirectional BFS will succeed for any positive reachability query.

DAG-Decomposition

Since a DAG G always contains at least one sink and one source we can decompose a DAG completely by deleting sources and sinks iteratively from G. In general, a DAG has several sources and sinks. To induce an order on the nodes to be deleted we define a priority function (see Section 4.1.1 for details). Using a generic priority function we take the sink or source v with the lowest priority in each step. Then the incident edges to v are added to either E_{fwd} if v was a source or their according reverse edges to E_{bwd} if v was a sink. Subsequently we delete v from G and update the priority of all affected nodes. We repeat this until only one node remains. Thus every edge has been assigned to E_{fwd} or E_{bwd} and $E_{fwd} \cup \overline{E_{bwd}} = E$. Figure 4.1 in Section 4.1 provides an example of such a DAG decomposition.

Following this decomposition scheme for an edge $e = (u, v) \in E_{fwd}$ the node u was deleted as a source and for an edge $e = (u, v) \in E_{bwd}$ the node v was deleted as a sink.



(a) DAG G with 2 edges

(b) Invalid search spaces that cannot be constructed using DAG-Decomposition on G

Figure 3.1: A simple base case for Lemma 2

As we want to restrict the forward search on E_{fwd} and the backward search on E_{bwd} it is necessary that for any nodes s, t with $s \to t$ there exists a node v, such that there exists an (s, v)-path in $G(E_{fwd})$ and there exists an (t, v)-path in $G(E_{bwd})$.

Lemma 1. *Let $G = (V, E)$ be a DAG and E_{fwd}, E_{bwd} the forward and backward search spaces obtained by a DAG-Decomposition. For any node $v \in V$ the following holds:*

$$\exists(v, u) \in E_{bwd} \implies \neg\exists(v, w) \in E_{fwd}$$

Intuitively this means, if any node v has an outgoing edge that has been added to E_{bwd}, no outgoing edge of v has been added to E_{fwd}.

Proof. Suppose there exists a node v and $\exists(v, u) \in E_{bwd} \wedge \exists(v, w) \in E_{fwd}$, Figure 3.1 depicts the most simple case. Then $(v, u) \in E_{bwd}$ implies that v was deleted as a sink. However this contradicts $(v, w) \in E_{fwd}$ which requires that v was deleted as a source. Hence $\exists(v, u) \in E_{bwd} \implies \neg\exists(v, w) \in E_{fwd}$ □

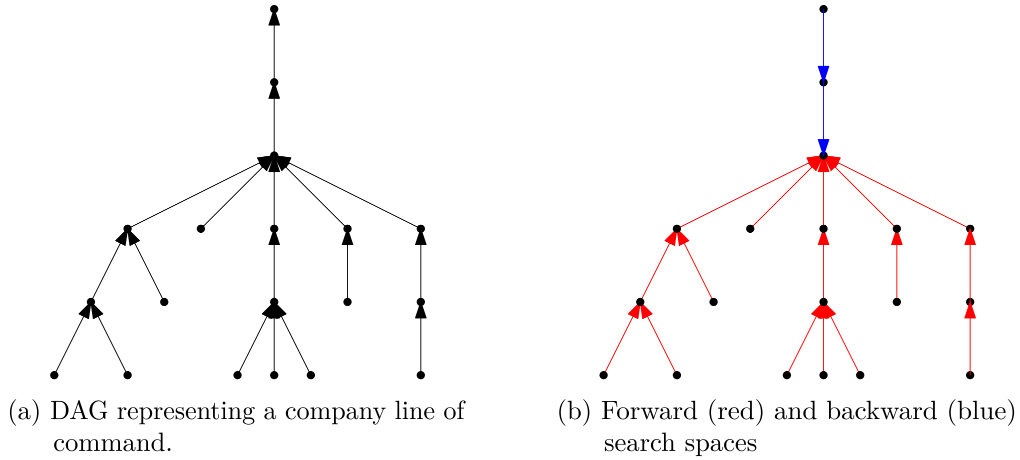
Lemma 2. *Let $G = (V, E)$ be a DAG and E_{fwd}, E_{bwd} the forward and backward search spaces obtained by a DAG-Decomposition. For any path $P = (v_1, \ldots, v_k)$ in G there exists a $d \in \{1, \ldots, k\}$, such that $\bigcup_{i<d}(v_i, v_{i+1}) \subseteq E_{fwd}$ and $\bigcup_{i\geq d}(v_{i+1}, v_i) \subseteq E_{bwd}$.*

Proof. We distinguish two cases. As for every edge $(v_i, v_{i+1}) \in E$ either $(v_i, v_{i+1}) \in E_{fwd}$ or $(v_{i+1}, v_i) \in E_{bwd}$ holds, either $(v_1, v_2) \in E_{fwd}$ or $(v_2, v_1) \in E_{bwd}$ holds.

Assume $(v_2, v_1) \in E_{bwd}$: If $k = 2$, then $d = 1$. Otherwise, suppose the set $F := \{j \in \{2, \ldots, k-1\} \mid (v_j, v_{j+1}) \in E_{fwd}\}$ is not empty. Hence, there exists a $j = \min(F)$. Then, $(v_j, v_j - 1) \in E_{bwd} \wedge (v_j, v_{j+1}) \in E_{fwd}$ holds, but contradicts with Lemma 1. Therefore $F = \emptyset$ and $\bigcup_{1\leq i<k}(v_{i+1}, v_i) \subseteq E_{bwd}$ holds. And therefore, $d = 1$.

Now assume that, $(v_1, v_2) \in E_{fwd}$. If $k = 2$, then $d = k = 2$. Otherwise, let $F := \{j \in \{2, \ldots, k-1\} \mid (v_{j+1}, v_j) \in E_{bwd}\}$. If $F = \emptyset$, then $\bigcup_{1\leq i<k}(v_i, v_{i+1}) \subseteq E_{fwd}$ holds and therefore, $d = k$. If $F \neq \emptyset$, let $j = \min(F)$. Then, $(v_{j-1}, v_j) \in E_{fwd}, (v_{j+1}, v_j) \in E_{bwd}$ holds. Because of $(v_{j+1}, v_j) \in E_{bwd}$, we can apply the first case to the subpath $(v_j, \ldots, v_k)$, which yields $\bigcup_{j\leq i<k}(v_{i+1}, v_i) \subseteq E_{bwd}$. And therefore, $d = j$. □

Since for a positive s, t query there exists at least one (s, t)-path, Lemma 2 shows that for such an (s, t)-path there exists a node v, such that in $G(E_{fwd})$ there exists an (s, v)-path and in $G(E_{bwd})$ there exists an (t, v)-path.



(a) DAG representing a company line of command.

(b) Forward (red) and backward (blue) search spaces

Figure 3.2: Example for search spaces using a simple DAG

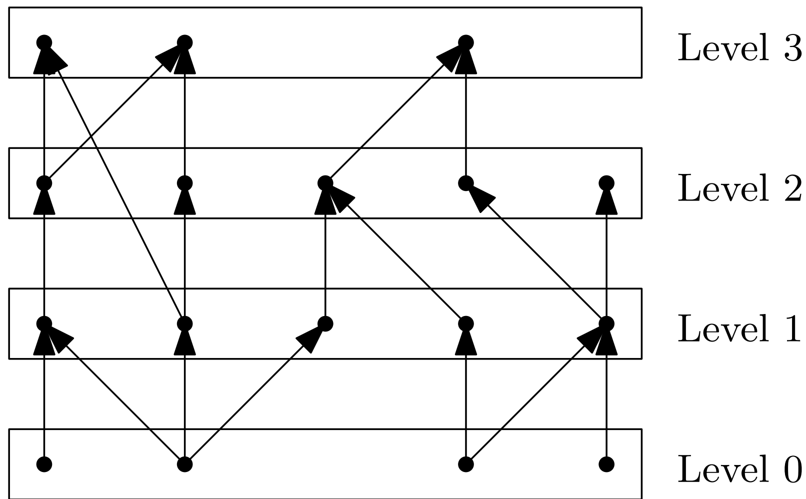


Figure 3.3: Topological Levels of a DAG G

The motivation behind splitting the search space in two is to stop a search if the branching factor gets high. A priority function which prefers nodes with a low degree can increase the number of deleted edges incident to a source or sink of height degree before that node is being deleted itself and therefore reduce the branching factor. A simple example is a graph representing the hierarchy in a company, see Figure 3.2. In that case the resulting search spaces end at the node with the highest degree.

3.2 Topological Levels

Chen *et al.* [6] introduced *topological levels* which we use to prune the search space. Given a DAG $G = (V, E)$ we define the topological level $L_G(v)$ by:

- $L(v) = 0$, if $d_G^-(v) = 0$
- $L(v) = \max(\{L(u) \mid (u, v) \in E\}) + 1$ else

We denote the number of levels $|\{L(v) : v \in V\}|$ by $l(G)$. The important information concerning reachability is the following Lemma.

Lemma 3. *Given a DAG G and two nodes $u, v \in V(G), u \neq v$ and the topological levels L_G. If $L_G(u) \geq L_G(v)$ then $u \not\to v$ holds.*

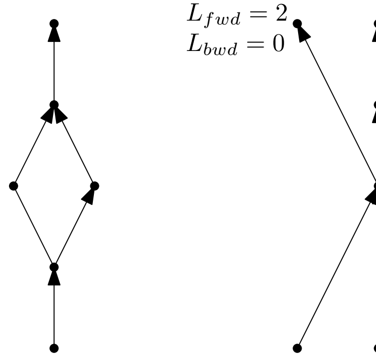


Figure 3.4: Left: DAG where $L_{bwd}(v) = l(G) - L_{fwd}(v) - 1$. Right: General case where the former doesn't apply

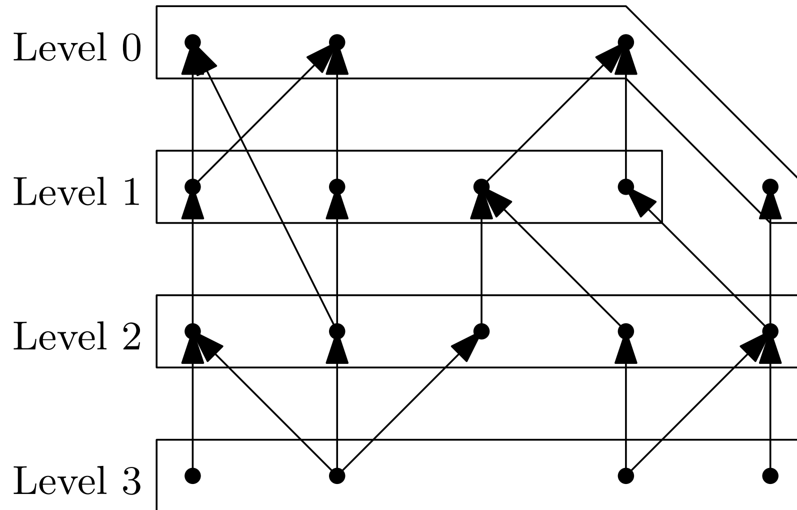
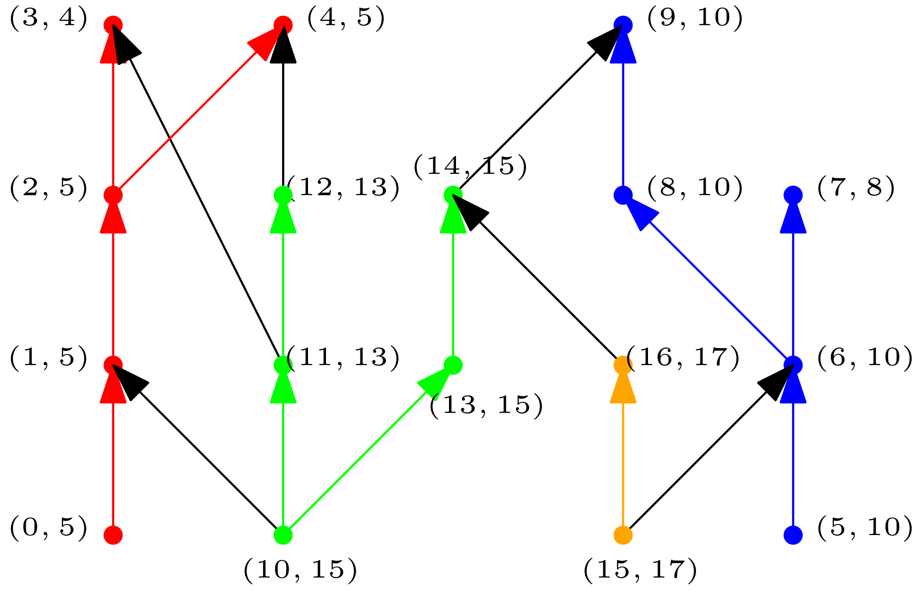


Figure 3.5: Topological Levels of $\overline{G}$, see Figure 3.3

Proof. Assume $L_G(u) \geq L_G(v)$ and $u \to v$. Then there exists a (u, v)-path $P = (w_1, \ldots, w_k)$ with $u = w_1$ and $v = w_k, k > 1$. For any $w_i, w_{i+1} \in P$, $L(w_{i+1}) \geq L(w_i) + 1$ holds, because $(w_i, w_{i+1}) \in E(G)$ and $L(w_{i+1}) = \max(\{L(u) \mid (u, w_{i+1}) \in E\}) + 1$. Therefore $L(v) = L(w_k) > L(w_{k-1}) > \cdots > L(w_1) = L(u)$ holds, which contradicts $L_G(u) \geq L_G(v)$. □

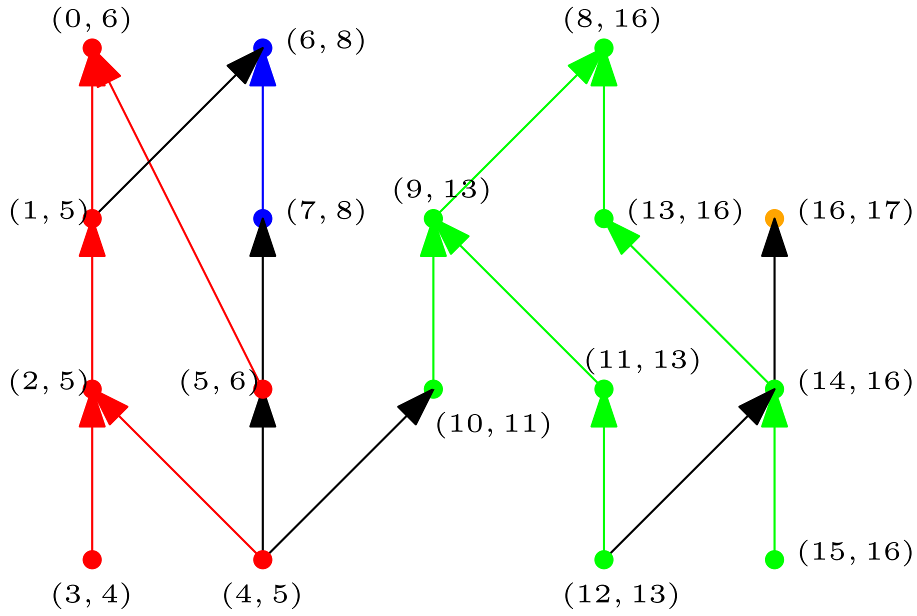
Topological levels are derived from topological sorting 2.2 but provide more information on reachability. Suppose $s : V(G) \to \mathbb{N}$ is a topological sorting on a DAG G and L_G are the topological levels of G. According to Lemma 3, for any two nodes $u, v \in V(G), u \neq v$ with $L_(u) = L(v)$, $u \not\to v$ and $u \not\to v$ both hold. W.l.o.g., let $s(u) < s(v)$, then we know that, $v \not\to u$ holds, but we have no information whether $u \not\to v$ holds too. For an example of topological levels of a DAG, see Figure 3.3.

Now we can define the *forward topological levels* L_G^{fwd} and the *backward topological levels* L_G^{bwd} , as $L_G^{fwd} = L_G$ and $L_G^{bwd} = L_{\overline{G}}$. Note that in some cases $L_G^{bwd}(v) = l(G) - L_G^{fwd}(v) - 1$ may hold, but in general this is not the case, see Figure 3.4. In Figure 3.5 we show L_G^{bwd} for the DAG G of Figure 3.3.



(10, 15)

(a) A DAG G with colored $DT_{fwd}(G)$



(b) A DAG G with colored $DT_{bwd}(G)$

Figure 3.6: Example of DFS-Trees, normal 3.6a and reverse 3.6b

3.3 DFS-Trees

We introduce *DFS-Trees* $DT(G)$ on a DAG G. $DT(G) = (DT_1, \ldots, DT_k)$ is an ordered partition of $V(G)$ where each $DT_i \in DT(G)$ induces a sub-DAG in G containing only one source $r(DT_i)$. Furthermore, $G(DT_1)$ is the maximal DAG in G containing $r(DT_1)$ as a sole source and each $G(DT_i)$ is maximal in $G - \bigcup_{j<i} DT_j$ containing $r(D_i)$ as its sole source.

Lemma 4. *Given a DAG G and any source s of G. Let $V' \subseteq V(G)$ be the set containing all nodes a DFS, rooted at s, marked during traversal. Then $G' = G(V')$ is the maximal sub-DAG of G, that has s as a sole source.*

Proof. $G' = G(V')$ is maximal in respect of s being its sole source, if $V' = \{v \in V \mid s \to v\}$ holds. Therefore, it suffices to show that a DFS rooted at s marks exactly the nodes reachable from v. As a DFS only traverses forward-edges, it can only mark reachable nodes.

Suppose there exists a node $v \in V(G)$, with $v \notin V'$ and $s \to v$. Then, there exists an (s, v)-path P in G, with $P = (w_1, \ldots, w_k), w_1 = s, w_k = v$. As s has been marked by the DFS, there exists a $w_i \in P$, with $i = \min(\{j \mid w_j \text{ has not been marked}\})$.

Thus, w_{i-1} has been marked and there exists an edge $(w_{i-1}, w_i) \in E(G)$. Therefore, the call of 2.2 on w_{i-1} would have called 2.2 on w_i. However, this contradicts with w_i not being marked. Hence, there exists no node $v \in V(G)$, with $v \notin V'$ and $s \to v$ and G' is maximal in respect of s being its sole source. □

As of Lemma 4, a maximal sub-DAG G' of a DAG G can be easily constructed. We start a DFS on any source s of G and add all nodes that have been marked to a node set V'. Furthermore, $G - G'$ yields no additional sources compared to G. More precisely, for any source $v \in V(G - G')$, $d^-_G(v) = 0$ holds. Otherwise, there would exist an edge $(u, v), u \in V(G')$, which contradicts with Lemma 4. Therefore, we construct DT_i by running a DFS rooted at $r(DT_i)$ on $G - \bigcup_{j<i} DT_j$.

Given $DT(G)$ we define the label functions $tl^{order} : V(G) \to \{0, \ldots, n-1\}$ and $tl^{till} : V(G) \to \{1, \ldots, n\}$. Additionally we define $tl(v) = (tl^{order}(v), tl^{till}(v))$.

Algorithm 3.1: SetDFSNums(v, *dfsPos*)

```
  Data: DAG DT_i
1 begin
2   mark v
3   dfsNum(v) ← dfsPos
4   dfsPos ← dfsPos + 1
5   for (v, w) ∈ E(DT_i) do
6     if w is not marked then
7       DFS (w, dfsPos)
8   dfsNumMax(v) ← dfsPos
9   return dfsPos
```

For each $DT_i \in DT(G)$ we start Algorithm 3.1 with $v = r(DT_i)$, $dfsPos = 0$, visiting only nodes in DT_i. During such a DFS we set the following two labels:

- for $v \in DT_i : dfsNum(v) = dfsPos$ when first visiting v and
- for $v \in DT_i : dfsNumMax(v) = dfsPos$ when last visiting v before returning to its predecessor.

Let $o_i = |\bigcup_{j<i} DT_i|$. Then for $v \in DT_i$ we define $tl^{order}(v) = o_i + dfsNum(v)$ and $tl^{till}(v) = o_i + dfsNumMax(v)$.

Note that tl defines a tree on each $DT_i \in DT(G)$. Given this labeling scheme tl^{order} is a bijection. Further $tl^{till}(r(DT_i)) - tl^{order}(r(DT_i)) = |DT_i|$ holds. And $tl^{order}(r(DT_i)) = tl^{till}(r(DT_{i-1})$ for $0 < i \leq k$ holds. Additionally $tl^{order}(r(DT_i)) \leq tl^{order}(v) < tl^{till}(v) \leq tl^{till}(r(DT_i)$ holds for all $v \in DT_i$.

Lemma 5. *Given a DAG G, $DT(G)$ and an according tl. For any two nodes $u, v \in V(G)$, if $tl^{till}(u) > tl^{order}(v) \geq tl^{order}(u)$ then $u \rightarrow v$ holds.*

Proof. As $tl^{till}(u) > tl^{order}(v) \geq tl^{order}(u)$ holds, v and u are part of the same DFS-Tree DT_i. And since $tl^{order}(v) \geq tl^{order}(u)$ holds, u has been visited for the first time before v has been visited for the first time. Furthermore, as $tl^{till}(u) > tl^{order}(v)$ holds, v has been visited before u has been visited for the last time. Therefore v has been visited by a recursion of the DFS that stared at u. Therefore, there exists a tree $T \subseteq G(DT_i)$ with u as its root, hence an (u, v)-path exists. □

Lemma 6. *Given a DAG G, $DT(G)$ and an according tl. For any two nodes $u, v \in V(G)$, if $tl^{till}(v) \leq tl^{order}(u)$ then $v \not\rightarrow u$ holds.*

Proof. Either $\exists DT_i \in DT(G) : u, v \in DT_i$ or $u \in DT_i$ and $v \in DT_j$ for $i \neq j$. If the former holds and if $tl^{till}(v) \leq tl^{order}(u)$ holds too, u was visited for the first time by the DFS after v was visited for the last time. In that case there exists no tree $T \subseteq G(DT_i)$ with v as root that contains u and there for there exists no (v, u)-path in $G(DT_i)$. And as $G(DT_1)$ is the maximal DAG in G with $r(DT_1)$ as sole source and each $G(DT_i)$ is maximal in $G - \bigcup_{j<i} DT_j$, there exists no (v, u)-path in G.

Otherwise, if $u \in DT_i$ and $v \in DT_j$ for $i \neq j$ holds. Then, since $tl^{till}(v) \leq tl^{order}(u)$ holds, $j < i$ holds, too. Thus, there exists no (v, u)-path in G, otherwise u would be contained in DT_j since $G(DT_j)$ is maximal in $G - \bigcup_{l<j} DT_l$. □

Furthermore, we define $DT_{bwd}(G) = DT(\overline{G})$, which we call *backward DFS-Tree* and therefore we call $DT_{fwd}(G) = DT(G)$ the *forward DFS-Tree*, see Figure 3.6. We use the notation tl_{fwd} and tl_{bwd}, respectively.

3.4 Base Reachability Query

This section introduces the basic idea of the query algorithm using a bidirectional BFS and the pruning methods we presented. We omit several aspects regarding performance in order to keep the algorithm simple and focus on the main idea of the query algorithm. Later we discuss several improvements in Section 3.5.

Given a DAG $G = (V, E)$, the forward and backward search spaces E_{fwd} and E_{bwd}, the topological levels L_{fwd} and reverse topological levels L_{bwd}, and the DFS-Tree labels tl_{fwd} and tl_{bwd}, we can use Algorithm 3.2 as query algorithm to answer a reachability query for $s, t \in V$ on G. It uses Algorithm 3.3 and Algorithm 3.4 to traverse G. Since we use a bidirectional BFS we need to maintain two first-in-first-out queues.

Algorithm 3.2 starts by pushing s into the forward queue and t into the backward queue then it repeatedly alternately calls Algorithm 3.3 and Algorithm 3.4 on the the popped node of their respective queue until either both queues run empty or the searches have met.

Algorithm 3.3 first checks if the active node v has been visited by the backward search, if so it sets *meet* to *true* and returns. In that case v has been visited by both search directions and a path from s to v exists in $G(E_{fwd})$ and a path from t to v exists in $G(E_{bwd})$ and therefore an (s, t)-path exists in G, according to Lemma 2. Next we check if $L_{fwd}(v)$ is

Algorithm 3.2: BaseQuery(s, t)

input : source node s, target node t
output : Boolean representing if $s \rightarrow t$
Data: G, E_{fwd}, E_{bwd}, L_{fwd}, L_{bwd}, tl_{fwd}, tl_{bwd}

```
1  begin
2      forward ← true
3      meet ← false
4      PushToForwardFifo(s)
5      PushToBackwardFifo(t)
6      while ¬meet and ¬(ForwardFifoIsEmpty() and BackwardFifoIsEmpty())
       do
7          if forward or BackwardFifoIsEmpty() then
8              v ← PopFromForwardFifo()
9              ForwardBFS(v)
10             forward ← false
11         else
12             v ← PopFromBackwardFifo()
13             BackwardBFS(v)
14             forward ← true
15     return meet
```

Algorithm 3.3: ForwardBFS(v)

input : node v
Data: G, L_{fwd}, tl_{fwd}, $meet$, t

```
1  begin
2      MarkForward(v)
3      if IsMarkedBackward(v) then
4          meet ← true
5          return
6      if L_fwd(v) ≥ L_fwd(t) then return
7      if tl_fwd^order(t) ≥ tl_fwd^till(v) then return
8      if tl_fwd^till(v) > tl_fwd^order(t) ≥ tl_fwd^order(v) then
9          meet ← true
10         return
11     for u ∈ N_G^+(v) and ¬IsMarkedForward(u) do
12         if (v, u) ∈ E_fwd then PushToForwardFifo(u)
```

Algorithm 3.4: BackwardBFS(v)

```
    input  : node v
    Data: G, L_bwd, tl_bwd, meet, s
 1  begin
 2      MarkFBackward(v)
 3      if IsMarkedForward(v) then
 4          meet ← true
 5          return
 6      if L_bwd(v) ≥ L_bwd(s) then return
 7      if tl_bwd^order(s) ≥ tl_bwd^till(v) then return
 8      if tl_bwd^till(v) > tl_bwd^order(s) ≥ tl_bwd^order(v) then
 9          meet ← true
10          return
11      for u ∈ N_G^-(v) and ¬IsMarkedBackward(u) do
12          if (v, u) ∈ E_bwd then PushToBackwardFifo(u)
```

greater or equal than $L_{fwd}(t)$, if that's the case we can prune the search space because the former holds for every node in the branch starting at v according to Lemma 3. Also if $tl_{fwd}^{order}(t) \geq tl_{fwd}^{till}(v)$ we can prune according to Lemma 6. In both cases we can return instantly. Furthermore, we check if $tl_{fwd}^{till}(v) > tl_{fwd}^{order}(t) \geq tl_{fwd}^{order}(v)$, if that is the case, $s \to t$ holds according to Lemma 5. Thus, we set *meet* to *true* and return. Now we check each node u in $N_G^+(v)$ that has not yet been marked by the forward search and if the edge incident to u and v is in E_{fwd} we push u into the according queue.

Algorithm 3.4 proceeds basically the same way. First it checks if the active node v has been visited by the forward search, if so it sets *meet* to *true* and returns. Next we check whether $L_{bwd}(v)$ is greater or equal than $L_{bwd}(s)$, if that's the case we can prune the search space. Also, if $tl_{bwd}^{order}(s) \geq tl_{bwd}^{till}(v)$ we can prune at v. As for the forward search in both cases we return. Furthermore, we check if $tl_{bwd}^{till}(v) > tl_{fwd}^{order}(s) \geq tl_{bwd}^{order}(v)$, if that is the case, we know $s \to t$ and we can set *meet* to *true* and return. Then we check each node u in $N_G^-(v)$ that has not yet been marked by the by the backward search and, if the edge incident to u and v is in E_{bwd}, we push u into the according queue.

3.5 More Pruning

In this section we introduce methods using additional auxiliary data and improvements based on the previous techniques. In Section 4.2 we implement the methods in an improved version of Algorithm 3.2 and in Section 5.2 we finally compare the impact of our pruning and shortcut techniques.

3.5.1 Reverse Topological Levels

In Algorithm 3.3 we can also check for pruning via reverse topological levels. If $L_G^{bwd}(v) \leq L_G^{bwd}(t)$ we can prune at v according to Lemma 3. We can use L_G^{fwd} in Algorithm 3.4, respectively.

3.5.2 Peek Nodes

Given the DFS-Trees $DT(G)$ and a node $v \in V(G)$ we can calculate *peek nodes* for v. In general a peek node p of a node v is a node that is reachable from v. We define three types of peek nodes:

- *minimal* peek nodes $pmin(v)$: a peek node p which minimizes

$$\{tl^{order}(p) \mid p \in V(G), v \to p\} \text{ and } tl^{order}(p) \leq tl^{order}(v)$$

- *tree* peek nodes $ptree(v)$: a peek node p which maximizes

$$\{tl^{till}(p) - tl^{order}(p) \mid p \in V(G), v \to p\} \text{ and } tl^{order}(p) < tl^{order}(v)$$

- *maximal* peek nodes $pmax(v)$: a peek node p which maximizes

$$\{tl^{till}(p) \mid p \in N^+(v), tl^{order}(p) < tl^{order}(v)\} \cup \{tl^{till}(p) \mid p = pmax(w), w \in N^+(v)\}$$

Intuitively, the minimal peek of a node v is the node p with the minimal tl^{order} reachable from v. The tree peek node of a node v is the node $p \neq v$ with the largest sub-DFS-Tree reachable from v.

The definition of a maximal peek node p of a node v is more complex. Therefore, we will explain the two sets it maximizes in detail. Let $P_1 = \{p \in N^+(v) \mid tl^{order}(p) < tl^{order}(v)\}$. P_1 contains all out-neighbors of v that have been visited before v during the construction of tl. Hence, they are either part of a previous DFS-Tree or a previous branch of the same DFS-Tree as v. Suppose $P_1 = N^+(v)$, then v is last node of its branch during construction of the DFS-Tree containing v. Thus, for any node u reachable from v, there exists a $w \in P_1$, such that $w \to u$ holds. According to Lemma 6, $tl^{order}(u) < tl^{till}(w)$ holds. Therefore, for $k = \max(\{tl^{till}(p) \mid p \in P_1\})$, k is larger than the maximal tl^{order} reachable from v. Hence, any node with an tl^{order} between k and $tl^{order}(v)$ is not reachable from v.

Now, let $P_2 = \{p \mid p = pmax(w), w \in N^+(v)\}$. Note, that for any node $p \in P_1$, $tl^{till}(p) > tl^{till}(pmax(p))$ holds. Since later, we choose the node that maximizes $\{tl^{till}(p) \mid p \in P_1 \cup P_2\}$, we only need to consider nodes in $P_3 = \{p \mid p = pmax(w), w \in (N^+(v) \setminus P_1)\}$. P_3 contains the maximal peek nodes of all successors of v visited after v during construction of tl. We use the previous definition $k = \max(\{tl^{till}(p) \mid p \in P_1\})$. If $k' \geq k$, for $k' = \max(\{tl^{till}(p) \mid p \in P_3\})$, is smaller than $tl^{order}(v)$, then any node with an tl^{order} between k' and $tl^{order}(v)$ is not reachable from v. Otherwise, if $tl^{order}(v) < k'$, we cannot use k' to predict the reachability of a node with a tl^{order} smaller than k'.

Intuitively, we can use a maximal peek node of a node v, to check if a node u, is placed between the node with the maximal tl^{order}, that is lower than $tl^{order}(v)$, still reachable from v and v. In Figure 3.7 we show a DAG G with according tl^{order} labels. The node v with $tl^{order}(v) = 13$ has the maximal peek node u ($tl^{order}(u) = 6$). Therefore, $tl^{till}(pmax(v)) = tl^{till}(u) = 9$. Thus, the red colored nodes cannot be reached from v, as their tl^{order} ranges from 9 to 12.

Note, that in case a peek node of a node v was inherited from an out-neighbor u ($pmax(v) = pmax(u)$), it is possible that $tl^{till}(pmax(v)) > tl^{order}(v)$ holds. This behavior is intended, since it allows faster construction and does not interfere with Lemma 9. Following Lemma 9, we explain how we can use maximal peek nodes during the BFS to prune and proof it.

In Figure 3.8 we show an example of a DAG G an according DFS-Tree labeling tl. Since $tl^{order}(v) = 13$ we set the minimal peek node to u, because $tl^{order}(u) = 2$ is the node

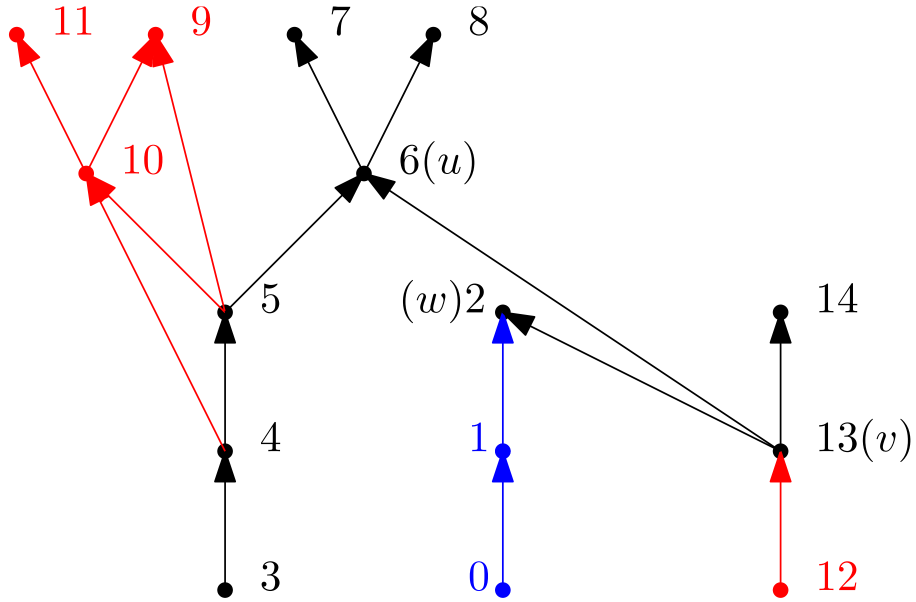
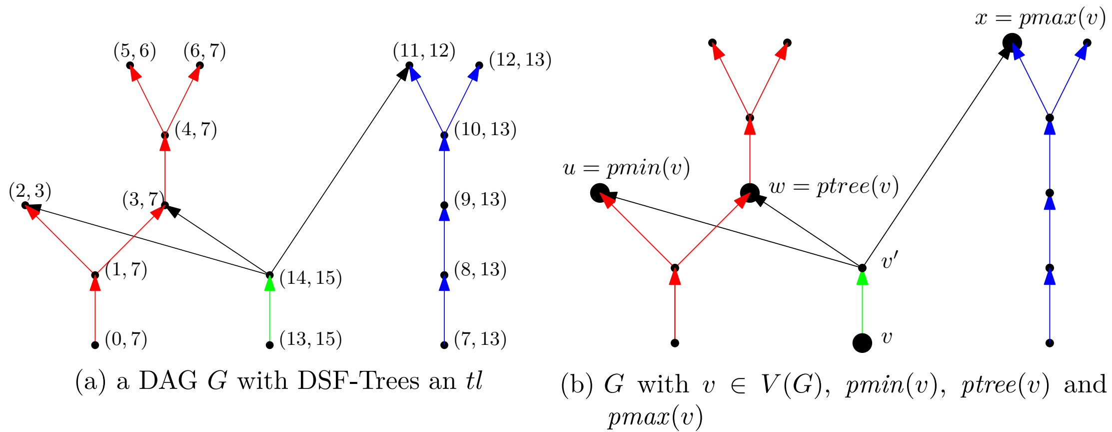


Figure 3.7: A DAG G labeled with tl^{order}. Concerning node v, the red colored nodes are not reachable from v, according to Lemma 8, and the blue colored nodes are not reachable from v, according to Lemma 9.



(a) a DAG G with DSF-Trees an tl

(b) G with $v \in V(G)$, $pmin(v)$, $ptree(v)$ and $pmax(v)$

Figure 3.8: A peek nodes example

with minimal tl^{order} reachable from v. Furthermore, w is the node that has the maximal subtree and $tl^{order}(w) = 3 < 13$, hence it becomes $ptree(v)$. To determine the maximal peek node of v we have to know the maximal peek node v', first. We observe that v' has the out-neighbors u, w, x. Both, x and u have no outgoing edges, therefore $pmax(u)$ and $pmax(x)$ are ϵ (they do not exist). Likewise, w leads to a simple tree, which also results to $pmax(w) = \epsilon$. Therefore the maximal peek node of v' is x, since x is the out-neighbor of v' which maximal tl^{till}. Now we can focus on v. We observe that v has no out-neighbors of smaller tl^{order}. Additionally x maximizes $\{tl^{till}(p) \mid p = pmax(w), w \in N^+(v)\} \cup \{tl^{till}(p) \mid p \in N^+(v), tl^{order}(p) < tl^{order}(v)\}$. Hence, x is the maximal peek node v. Note that peek nodes of v can be part of the same DFS-Tree as v. Also $pmin(v) = ptree(v) = pmax(v)$ may apply some times.

Lemma 7. *Given a DAG G and according functions tl and $ptree$. For any two nodes $v, t \in V(G)$,*

$$tl^{till}(ptree(v)) > tl^{order}(t) \geq tl^{order}(ptree(v)) \implies v \to t$$

holds.

Proof. Assume $tl^{till}(ptree(v)) > tl^{order}(t) \geq tl^{order}(ptree(v))$ holds. By definition of $ptree$, $v \to ptree(v)$ holds. Additionally, Lemma 5 holds for the tow nodes $ptree(v)$ and t. Which yields $ptree(v) \to t$. Therefore, $v \to t$ holds. □

According to Lemma 7 we can check in each call of Algorithm 3.3 on v if $tl^{till}(ptree(v)) > tl^{order}(t) \geq tl^{order}(ptree(v))$, if so, we can answer the query positively.

Lemma 8. *Given a DAG G an according functions tl^{order} and $pmin$. For any two nodes $v, t \in V(G)$,*

$$tl^{order}(pmin(v)) > tl^{order}(t) \implies v \not\to t$$

holds.

Proof. Assume $tl^{order}(pmin(v)) > tl^{order}(t)$ holds. Since $pmin(v)$ is the node with the smallest tl^{order} reachable from v and $tl^{order}(t)$ is smaller, $v \not\to t$ holds. □

Additionally, we can check in each call of Algorithm 3.3 on v if $tl^{order}(pmin(v)) < tl^{order}(t)$. If the former holds, we can prune according to Lemma 8.

Lemma 9. *Given a DAG G an according functions tl and $pmax$. For any two nodes $v, t \in V(G)$,*

$$tl^{till}(pmax(v)) \leq tl^{order}(t) < tl^{order}(v) \implies v \not\to t$$

holds.

Proof. We defined $pmax$ the way that for any node u, if $tl^{order}(pmax(u)) \leq tl^{order}(u)$ holds, $tl^{till}(pmax(u)) > tl^{order}(w)$ holds for any node w, $tl^{order}(w) \leq tl^{order}(u)$. Intuitively, $tl^{till}(pmax(u)) - 1$ is the largest order of a node x, that is smaller than the order of u and still reachable by u. Hence, any node x with a $tl^{order}(x)$ between $tl^{till}(pmax(u))$ (including) and $tl^{order}(u)$ (excluding) is not reachable from u.

Assume $tl^{till}(pmax(v)) \leq tl^{order}(t) < tl^{order}(v)$ holds. Then $v \not\to t$ holds by definition of $pmax$, since $tl^{till}(pmax(v)) < tl^{order}(v)$ holds. □

According to Lemma 9 we can check in each call of Algorithm 3.3 on v if $tl^{till}(pmax(v)) \leq tl^{order}(t) < tl^{order}(v)$ holds. If it does hold, we can prune the search space at v.

We could also use $pmin(v)$ and $pmax(v)$ to check for positive reachability, but as those checks are expensive and by discarding that option we can save index size and lookups by just storing $tl^{order}(pmin(v))$ and $tl^{till}(pmax(v))$ we decided not to use them in our algorithm.

Peek nodes can be used on DT_{fwd} and DT_{bwd}, hence the same applies for Algorithm 3.4 and we only have to substitute t by s for the checks. Furthermore, we can calculate peek nodes efficiently while constructing the DFS-Trees as shown in Section 4.1.3.

4. Construction and Query

4.1 Construction

The Search Spaces (3.1), DFS-Trees (3.3), Peek Nodes (3.5.2) and Topological Levels (3.2) for a DAG G are calculated in a construction phase. DFS-Trees, Peek Nodes and Topological Levels can be stored as auxiliary data of the nodes $V(G)$, whereas the Search Spaces $E_{fwd}(G)$ and $E_{bwd}(G)$ replace $E(G)$ since $E_{fwd}(G) \cup \overline{E}_{bwd}(G) = E(G)$. The former three can be computed in linear time regarding nodes and edges and identifying the Search Spaces needs $O(m + n \log n)$ time if we use a general priority function. If we do not prioritize the sources and sinks we would only have to maintain candidates for deletion without any ordering, which needs $O(n + m)$ time.

4.1.1 Construction of Search Spaces

We can construct the Search Spaces E_{fwd} and E_{bwd} as described in Section 3.1 by iteratively deleting sources and sinks. At first we initialize E_{fwd} and E_{bwd} as empty sets and add edges to them during node deletion. To obtain the next node ready for deletion we maintain the candidates in a priority queue, which we initialize by adding all sources and sinks of the DAG G along with their calculated priority. We delete the node with the lowest priority and assign its incident edges to either E_{fwd} or E_{bwd} until only one node remains in the queue. When deleting a node we have to update the priority queue by adding all new sources and sinks. Thus at any time all sinks and sources of the current DAG are included in the priority queue. Finally, by Lemma 10, we know that if only one node remains in the priority queue, all edges have been assigned.

We use

$$prio(v) = \begin{cases} d_G(v) \text{ if } v \text{ is a sink or source in } G' \\ n \text{ else} \end{cases}$$

as a priority function, where G is the original graph and G' is the current graph missing the deleted nodes. It provides a low priority for nodes of small degree, thus it delays the deletion of high degree nodes.

Lemma 10. *Let $G = (V, E)$ be a DAG and $S = \{v \in V \mid d^-(v) = 0 \vee d^+(v) = 0\}$ then $|S| = 1 \implies E = \emptyset$ holds.*

Proof. Suppose $|S| = 1$ and $E \neq \emptyset$, then there exists an edge $e = (u, v) \in E$. Let p be a maximal path that contains e and let s be the start node and t the end node of p, since p is maximal and $(t, s) \notin E$ as G is acyclic, s is a source and t is a sink. Thus $d^-(s) = 0$ and $d^+(t) = 0$ and therefore $s, t \subseteq S$ which contradicts with $|S| = 1$, since $s \neq t$. □

We use Algorithm 4.1 to construct the Search Spaces E_{fwd} and E_{bwd} of a DAG G.

Algorithm 4.1: Construction of Search Spaces E_{fwd} and E_{bwd}

```
   input  : DAG G = (V, E)
   output : E_fwd, E_bwd
 1 begin
 2     E_fwd ← {}
 3     E_bwd ← {}
 4     PQ ← empty priority queue
       // initialize the priority queue with sinks and sources of G
 5     for v ∈ V do
 6         if v is a source of sink then
 7             priority ← prio(v)
 8             Push(PQ, v, priority)
       // delete sinks and sources according to their priority
 9     while Size(PQ) > 1 do
10         v ← DeleteMin(PQ)
11         DeleteNode(v)
```

Algorithm 4.2: DeleteNode(v)

```
   input  : node v
   Data: DAG G = (V, E), E_fwd, E_bwd, priority queue PQ
 1 begin
 2     if IsSource(v) then
 3         for u ∈ N+(v) do
 4             E_fwd ← E_fwd ∪ {(v, u)}
 5             E ← E \ {(v, u)}
               // check if u has become a source or sink, if so push it
                  into the queue
 6             priority ← NodePriority(u)
 7             if priority < |V| then
 8                 Push(PQ, u, priority)
 9     else
10         for u ∈ N−(v) do
11             E_bwd ← E_fwd ∪ {(u, v)}
12             E ← E \ {(u, v)}
               // check if u has become a source or sink, if so push it
                  into the queue
13             priority ← NodePriority(u)
14             if priority < |V| then
15                 Push(PQ, u, priority)
```

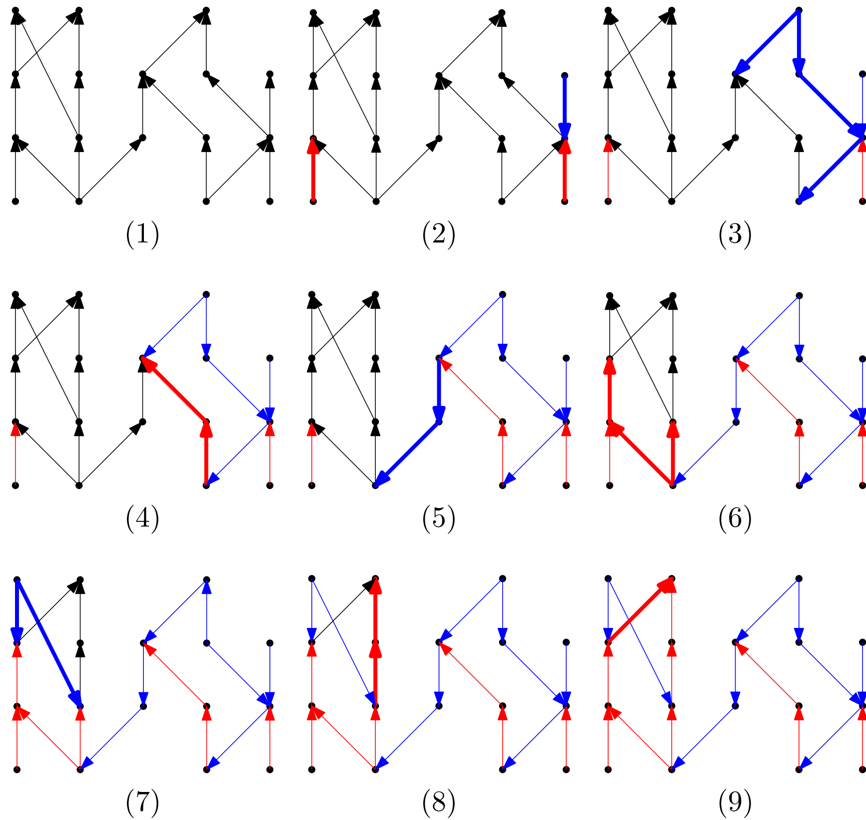


Figure 4.1: Building search spaces E_{fwd} (red) and E_{bwd} (blue) in a DAG G.
To simplify the process we handle nodes with only one edge left in one step.
Therefore, each figure except (7) and (9) contains multiple node deletions.

4.1.2 Setting Topological Levels

We use a modified DFS to compute the topological levels of a DAG G. For each node $v \in V(G)$ we maintain a counter $visted(v)$, which is initially set to zero. Furthermore, we initialize $L(v) = 0$ for any node v of G. We sequentially start the modified DFS on the sources of G. When we traverse an edge $(u, v) \in E(G)$ we update the topological level of v to $\max(\{L(v), L(u) + 1\})$ and increase $visited(v)$ by one. We only recurse at v if $visited(v)$ equals the $d^-_G(v)$.

Additionally, we count the number of nodes, on which we recursed during the modified DFS, started at a source s. We store this number as $TreeSize(s)$ and use it as a heuristic to determine an ordering of the sources used in Section 4.1.3. Algorithm 4.4 and Algorithm 4.3 implement such a modified DFS.

Algorithm 4.4 constructs the topological levels L for an input DAG G. More precisely, it constructs L_{fwd}. In order to obtain L_{bwd} we can just run Algorithm 4.4 on $\overline{G}$.

4.1.3 Construction of DFS-Trees

Algorithm 4.6 constructs DFS-Trees $DT(G)$ for a DAG G and also calculates $pmin(v)$, $ptree(v)$ and $pmax(v)$ for each node v in linear time regarding edge count. To determine the Topological Levels we start a DFS from each source of G. Each DFS only traverses edges that haven't been traversed yet by a former DFS.

Algorithm 4.3: SetLevel(v)

```
  input  : node v, size nsize
  output : size nsize
  Data: G, L, visited
1 begin
2   for (v, u) ∈ E(G) do
3     if L(u) ≤ L(v) then
4       L(u) ← L(v) + 1
5     visited(u) ← visited(u) + 1
6     if visited(u) = d⁻(u) then
7       nsize ← SetLevel (u, nsize + 1)
8   return nsize
```

Algorithm 4.4: SetLevels()

```
  input  : DAG G
  Data: L, TreeSize
1 begin
2   for s ∈ S_source(G) do
3     L(s) ← 0
4     TreeSize(s) ← SetLevel (s, 1)
```

Hence, the ordering of the sources in which we process the DFS has an impact on the number of the nodes in $DT(G)$. Using *TreeSize*, introduced in Section 4.1.2, we define such an ordering by O_{source}. Let $O_{source} = (s_1, \ldots, s_k)$ for $s_i \in S_{source}(G)$ for $k = |S_{source}(G)|$ and $TreeSize(s_i) \geq TreeSize(s_{i+1})$.

To build DT_1 we start a DFS on s_1. For each node $v \notin DT_1$ we visit, we:

- set $dfsNum(v)$,
- add v to DT_1,
- continue with its out-neighbors,
- set $dfsNumMax(v)$,
- set $tl^{order}(v) = dfsNum(v)$,
- set $tl^{till}(v) = dfsNumMax(v)$,
- set $pmin(v) = v$,
- set $ptree(v) = INVALID$,
- set $pmax(v) = INVALID$ and
- return to its predecessor we came from.

Then for each $s_i \in S_{source}, i \neq 1$ in the order given by O_{source} we build DT_i by starting a DFS on s_i. We start again with $dfsNum = 0$ and use the offset $o_i = |\bigcup_{j<i} DT_j| + 1$ to set the labels tl. For each node v we visit and which hasn't been visited yet we:

- set $dfsNum(v)$ as before,
- add v to DT_i

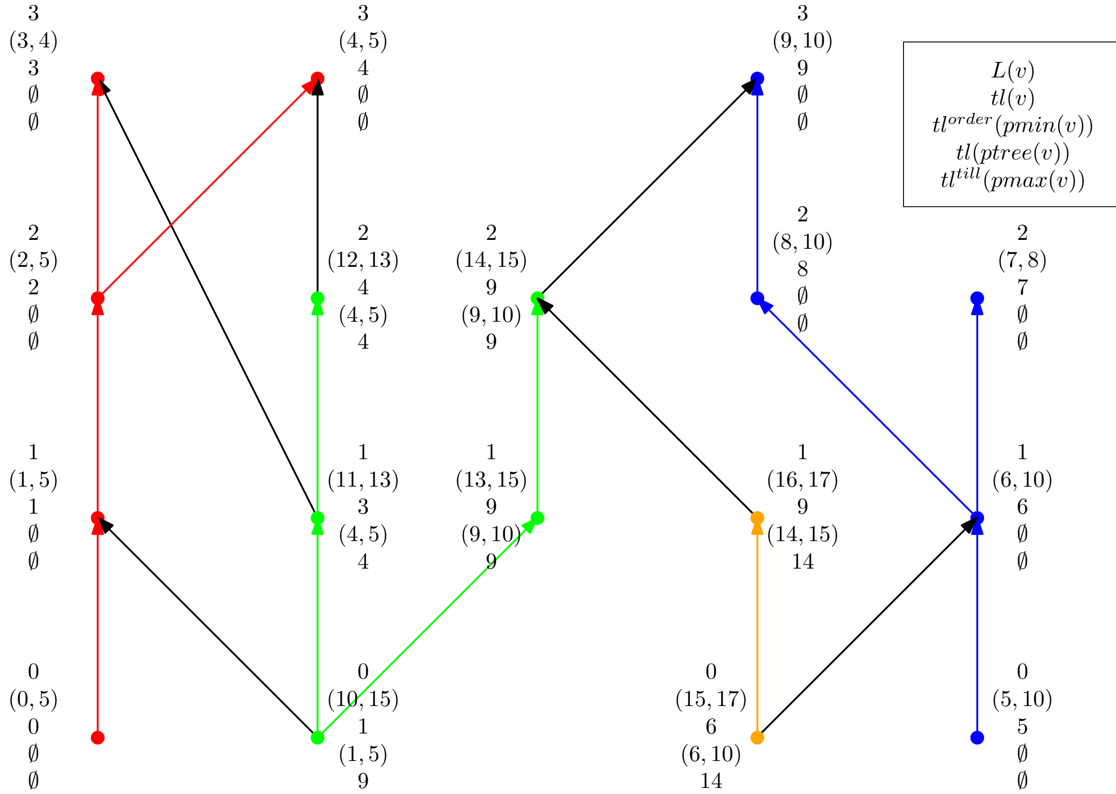


Figure 4.2: Forward DFS-Trees and according labels of a DAG G

- continue with its out-neighbors,
- set $dfsNumMax(v)$,
- set $tl^{order}(v) = o_i + dfsNum(v)$,
- set $tl^{till}(v)) = o_i + dfsNumMax(v)$,
- set $pmin(v)$ to the node u that minimizes $tl^{order}(u)$ for $u = pmin(w), w \in N_G^+(v)$, if none exist set $pmin(v)$ to v,
- set $ptree(v)$ to the node u that maximizes $tl^{till}(u) - tl^{order}(u)$ and is in one of the following two sets:
  - $\{u \in N^+(v) \mid tl^{order}(u) < tl^{order}(v)\}$ and
  - $\{ptree(u) \mid u \in N^+(v) \wedge ptree(u) \text{is set}\}$,

  if no such node exists we set $ptree(v)$ to $INVALID$,
- set $pmax(v)$ to the node u that maximizes $tl^{till}(u)$ for $u = pmax(w), w \in N_G^+(v)$ and $tl^{till}(u) \leq tl^{order}(v)$, if none exist set $pmax(v)$ to $INVALID$,
- return to the predecessor we came from.

Using this labeling scheme we obtain the labels $tl(v) = (tl^{order}(v), tl^{till}(v))$ and the peek nodes $pmin(v)$, $ptree(v)$ and $pmax(v)$ for each node $v \in V(G)$. Since the DFS-Tree DT_i is the maximal subtree in $G - \bigcup_{j<i} DT_j$ starting at r_i we can use Lemma 6 for pruning during the BFS.

Since we use minimal and maximal peek nodes only for pruning and tree peek nodes only for shortcuts we only need to store $tl^{peek_min}(v) := tl^{order}(pmin(v))$, $tl^{peek_max}(v) := tl^{till}(pmax(v))$, $tl^{peek_order}(v) := tl^{order}(ptree(v))$ and $tl^{peek_till}(v) := tl^{till}(ptree(v))$. This

is relevant for cache optimization and provides simpler labels for our figures, despite that we continue to refer to *pmin*, *ptree* and *pmax* throughout our algorithms, as they are more intuitive.

Algorithm 4.6 computes *tl*, *pmin*, *ptree* and *pmax* for a DAG G, as for topological levels this means it computes the forward DFS-Trees with tl_{fwd}, $pmin_{fwd}$, $ptree_{fwd}$ and $pmax_{fwd}$. We can compute the according backward DFS-Trees and therefore tl_{bwd}, $pmin_{bwd}$, $ptree_{bwd}$ and $pmax_{bwd}$ by running Algorithm 4.6 on $\overline{G}$.

Figure 4.2 shows the a DAG G with colored forward DFS-Trees and according labels including forward Topological Levels.

Algorithm 4.5: UpdatePeekNodes(v, u)

```
    input : node v, node u
    Data: pmin, pmax, ptree, tl^order, tl^till, tl
 1  begin
        // we define Range(x) := tl^till(x) − tl^oder(x)
 2      if ptree(u) is not invalid then
            // since v → u holds, check if u has a better ptree
 3          if Range (ptree(u)) > Range(ptree(v)) then
 4              ptree(v) ← ptree(u)
 5      if tl^order(u) < tl^order(v) then
            // u is part of an earlier branch or DFS-Tree.
            // Therefore, we have to check if u is an better ptree
 6          if Range(u) > Range(ptree(v)) then
 7              ptree(v) ← u
 8      if tl^order(pmin(u)) < tl^order(pmin(v)) then
            // we can reach a node with smaller tl^order.
            // Therefore, we update pmin
 9          pmin(v) ← pmin(u)
10      if pmax(u) is set then
            // check if we have to update pmax
11          if (pmax(v)is not set ∨ tl^till(pmax(u)) > tl^till(pmax(v))) then
                // In case pmax(v) is not set yet, we update it anyway.
                // Otherwise, we check pmax(u) is better
12              pmax(v) ← pmax(u)
```

Algorithm 4.6: SetDFSTrees(v, *dfsPos*)

```
    input : G, O_source
 1  begin
 2      dfsPos ← 0
 3      for s ∈ O_source do
 4          dfsPos ← SetDFSTree (s, dfsPos)
```

Algorithm 4.7: SetDFSTree(v, $dfsPos$)

```
    input  : node v, current dfsPos
    output : dfsPos
    Data: G, pmin, pmax, ptree, tl^order, tl^till, tl
 1  begin
 2      tl^order(v) ← dfsPos
 3      pmin(v) ← v
 4      ptree(v) ← invalid
 5      pmax(v) ← invalid
 6      dfsPos ← dfsPos + 1
        // iterate over all outgoing-neighbors of v
 7      for u ∈ N+(v) do
 8          if tl(u) is not set then
                // u has not been visited yet, therefore process u
 9              dfsPos ← SetDFSTree (u, dfsPos)
            // Since u has been processed in any case,
            // we have to update the peek nodes of v
10          UpdatePeekNodes(v, u)
11      tl^till(v) ← dfsPos
12      tl(v) ← (tl^order(v), tl^till(v))
13      return dfsPos
```

4.2 Query

Finally, we present our query algorithm. Given:

- a DAG G,
- E_{bwd} and E_{fwd}, obtained from Algorithm 4.1,
- L_{fwd} and L_{bwd}, obtained from Algorithm 4.4 and
- tl, $pmin$, $pmax$, $ptree$, obtained from Algorithm 4.6 for both directions.

We implement a query algorithm that performs a bidirectional BFS and utilizes Lemmata 3,6,8,9 for pruning and Lemmata 5,7 to shortcut during traversal.

Suppose, we visit a node v during the forward search of an s,t-query. There exists an (s,v)-path in $G(E_{fwd})$, as we traversed from s to v using only edges of E_{fwd}. Therefore, if v has been visited by the backward search, there also exists a (t,v)-path in $G(E_{bwd})$ and $s \to t$ holds according to Lemma 2. Furthermore, if $L_{fwd}(u) \geq L_{fwd}(t)$ or $L_{bwd}(t) \geq L_{bwd}(u)$ hold, $v \not\to t$ holds according to Lemma 6. Additionally, we use tl_{fwd}, $pmin_{fwd}$ and $pmax_{fwd}$ to check if $v \not\to t$ holds according to Lemma 6, Lemma 8 or Lemma 9. Finally we check if $v \to t$ holds, using tl_{fwd} and $ptree$ according to Lemma 5 and Lemma 7. If neither of the above holds, we add all out-neighbors of v in $G(E_{fwd})$, that have not been visited by the forward search, to the FIFO of the forward BFS, and continue the search. Otherwise, if $v \to t$ holds, there exists an (s,t)-path in G and we can answer the query positively. In case $v \not\to t$ holds, we prune the forward search space at v, since there exists no (v,t)-path.

During the backward search we proceed analogously, checking whether s is reachable from currently visited node v in $G(E_{bwd})$ or not. Algorithm 4.9 implements this step of the forward search. An according step of the backward search would be implemented by performing the checks against t instead of s and using the according bwd data.

Algorithm 4.8 implements such a query algorithm. Initially, we check if $s = t$, in which case $s \to t$ holds. Otherwise, we proceed with the bidirectional BFS, until the forward and backward searches meet, or both FIFOs run empty.

Algorithm 4.8: Query(s, t)

```
   input  : source node s, target node t
   output : Boolean representing if s → t
   Data: G
 1 begin
 2   if s = t then
 3     return true
     // start the bidirectional BFS
 4   forward ← true
 5   meet ← false
 6   PushToForwardFifo(s)
 7   PushToBackwardFifo(t)
 8   MarkForward(s)
 9   MarkBackward(t)
     // Loop until the searches have met, or both queue are empty
10   while ¬meet and ¬(ForwardFifoIsEmpty() and BackwardFifoIsEmpty())
     do
11     if forward or ¬BackwardFifoIsEmpty() then
12       v ← PopFromForwardFifo
13       ForwardBFS(v)
14       forward ← false
15     else
16       v ← PopFromBackwardFifo
17       BackwardBFS(v)
18       forward ← true
19   return meet
```

Algorithm 4.9: ForwardBFS(v)

```
    input : node v
    Data: G, L_fwd, tl_fwd, meet
 1  begin
        // Check if searches have met according to Lemma 2
 2      if IsMarkedBackward(v) then
 3          meet ← true
 4          return
        // Check for levels according to Lemma 3
 5      if L_fwd(v) ≥ L_fwd(t) then
 6          return
        // Check reverse Topological Levels according to Section 3.5.1
 7      if L_bwd(v) ≤ L_bwd(t) then
 8          return
        // Check if can prune via tl according to Lemma 6
 9      if tl^till_fwd(v) ≤ tl^order_fwd(t) then
10          return
        // Check if we can shortcut according to Lemma 5
11      if tl^till_fwd(v) > tl^order_fwd(t) ≥ tl^order_fwd(v) then
12          meet ← true
13          return
        // Check if we can prune using the minimal peek node according to
           Lemma 8
14      if tl^order_fwd(pmin_fwd(v)) > tl^order_fwd(t) then
15          return
        // Check for a short cut via the tree peek node according to
           Lemma 7
16      if tl^till_fwd(ptree_fwd(v)) > tl^order_fwd(t) ≥ tl^order_fwd(ptree_fwd(v)) then
17          meet ← true
18          return
        // Check if we can prune using the maximal peek node according to
           Lemma 9
19      if tl^till_fwd(pmax_fwd(v)) ≤ tl^order_fwd(t) < tl^order_fwd(v) then
20          return
21      for (v, u) ∈ E_fwd and ¬IsMarkedForward(u) do
22          MarkForward(u)
23          PushToForwardFifo(u)
```

5. Experiments

In this chapter we will first discuss the performance of P2REACH and the impact of the individual pruning options based on our experiments. Later we compare the performance of our approach with PATHTREE [16], GRAIL [34] and TF [6].

All of our experiments have been conducted on an Intel Xeon X5550 running at 2.67GHz with 8MB Level3 cache, 4 x 256kB Level2 cache and 48GB of DDR3 RAM. All algorithms have been implemented in C++ and have been compiled using gcc 4.8.2. The system ran Ubuntu 12.04.2 using a Linux kernel 3.5.

5.1 Test Data

Algorithm 5.1: GenerateRandomDAG

input : number of nodes n, number of edges m
output : DAG G

```
1  begin
2      V ← {0, ..., n − 1}
3      P ← permutation of V
4      i ← 0
5      while i < m do
6          s, t ← random pair with s, t ∈ V ∧ s ≠ t
7          if s < t then
8              E ← E ∪ {(P(s), P(t))}
9          else
10             E ← E ∪ {(P(t), P(s))}
11     return (V, E)
```

We provide experiments on graphs of five categories, mainly we use the same graphs used by [34], [16] and [6]:

Small Real Sparse: These graphs have edge-node ratio less than 1.2. They represent XML documents (`xmark, nasa`), metabolic networks (`amaze, kegg`), and the rest was introduces by the authors of GRAIL [34] and were obtained from BioCyc and represent pathway and genome databases. See Table 5.1a for a complete list and their properties.

Dataset	Nodes	Edges	$\lvert E\rvert/\lvert V\rvert$
agrocyc	12 684	13 657	1.07
amaze	3 710	3 947	1.06
anthra	12 499	13 327	1.07
ecoo	12 620	13 575	1.08
human	38 811	39 816	1.01
kegg	3 617	4 395	1.22
mtbrv	9 602	10 438	1.09
nasa	5 605	6 538	1.17
vchocyc	9 491	10 345	1.09
xmark	6 080	7 051	1.16

(a) small real sparse

Dataset	Nodes	Edges	$\lvert E\rvert/\lvert V\rvert$
arxiv	6 000	66 707	11.12
citeseer-sub	10 720	44 258	4.13
go	6 793	13 361	1.97
pubmed	9 000	40 028	4.45
yago	6 642	42 392	6.38

(b) small real dense

Dataset	Nodes	Edges	$\lvert E\rvert/\lvert V\rvert$
citeseer	693 947	312 282	0.45
citeseerx	6 540 399	15 011 259	2.30
cit-patents	3 774 768	16 518 947	4.38
go-uniprot	6 967 956	34 770 235	4.99
uniprot22m	1 595 444	1 595 442	1.00
uniprot100m	16 087 295	16 087 293	1.00
uniprot150m	25 037 600	25 037 598	1.00

(c) large real

Dataset	Nodes	Edges	$\lvert E\rvert/\lvert V\rvert$
rand1m2x	1M	2M	2
rand1m5x	1M	5M	5
rand1m10x	1M	10M	10
rand10m2x	10M	20M	2
rand10m5x	10M	50M	5
rand10m10x	10M	100M	10

(d) large random

Dataset	Nodes	Edges	$\lvert E\rvert/\lvert V\rvert$
email-EuAll	231 000	223 004	0.97
p2p-Gnutella31	48 438	55 349	1.15
soc-LiveJournal1	971 234	1 024 140	1.05
web-Google	371 764	517 805	1.39
wiki-Talk	2 281 879	2 311 570	1.01

(e) stanford

Table 5.1: Graphs used in our experiments

Small Real Dense: These graphs are mostly obtained from citation networks (`pubmed, citeseer, arxiv`). All of them have been initially used by [17].

Large Real: The authors of GRAIL [34] introduced seven large graphs to demonstrate their scaling abilities. `citeseer, citeseerx` and `cit-patents` are citation networks, `go-uniprot` is a joint graph of Gene Ontology terms and the annotations file from the UniProt database. The rest of the graphs (`uniprot22m, uniprot100m, uniprot150m`) are subsets of the UniProt RDF graph. See [34] for more detailed information on these graphs.

Large Random: These graphs are randomly generated DAGs. They where obtained by Algorithm 5.1 analogously to [34]. We use the naming scheme `rand`$\{N\}\{D\}$`x`, where N is the number of nodes (e.g.: `1m` for 1 million) and D is the average degree.

Stanford: These graphs were obtained from Stanford Large Network Dataset Collection. They were introduced by the authors of TF [6] and represent different real networks. These graph represent an email network from a EU research institution (`email-EuAll`), the Gnutella peer to peer network from August 31 2002 (`p2p-Gnutella31`), the LiveJournal online social network (`soc-LiveJournal1`), the Web graph from Google (`web-Google`) and the Wikipedia communication network (`wiki-Talk`).

All query times aggregate times for 100000 s, t reachability queries. We use three different types of such batch queries:

random: Random queries provided by randomly picking $s, t \in V(G), s \neq t$

positive: Positive queries obtained by randomly picking $s \in V(G), d_G(s) > 0$, calculating a tree T_G with s as root and then picking a random node $t \in T_G$.

negative: Negative queries obtained by randomly picking $s \in V(G)$, calculating a tree T_G with s as root and then picking a random node $t \notin T_G$.

5.2 Experiments on P2REACH

The techniques described in Chapter 3 can each provide a speedup for the BFS. In this section we demonstrate the impact of those techniques, by comparing results of experiments on various graphs using different subsets of *Search Spaces* (Section 3.1), *DFS-Trees* (Section 3.3) and *Topological Levels* (Section 3.2).

	Search Spaces	DFS-Trees	Topological Levels	Binary Priority
p2reach-sdl	x	x	x	
p2reach-s	x			
p2reach-				
p2reach-sd	x	x		
p2reach-sl	x		x	
p2reach-sdlb	x	x	x	x
p2reach-dl		x	x	

Table 5.2: Naming scheme for different P2REACH configurations

5.2.1 Search Spaces

As described in Section 3.1 our motivation was decreasing the branching factor during a BFS, therefore we compare *p2reach-*, without Search Spaces, with *p2reach-s* including Search Spaces. Both versions do not use any pruning or shortcut techniques. Therefore, they implement a normal bidirectional BFS, where the latter is restricted to the according Search Spaces.

In Table 5.3 we can see that Search Spaces provide a great speedup on nearly all graphs. For large random graphs, the speedup increases as they get denser and the branching factor of the normal bidirectional BFS gets higher. In case of the large real dataset, we observe that *p2reach-* can be slightly faster on negative queries, which have mostly a quite small branching factor on those graphs. But for those graphs *p2reach-s* is up to several orders of magnitude faster than *p2reach-*. Throughout the small datasets, Search Spaces provide a good speed up. Finally for the stanford dataset, using Search Spaces is at least two orders of magnitude faster than pure bidirectional BFS, on positive, as well as on negative queries. Only `p2p-Gnutella31` is an exception, but still Search Spaces provide a speed up of a factor of 3 on positive queries and nearly 100 on negative ones.

5.2.2 Priority Function for Search Spaces

In order to demonstrate the impact of the priority function used during the construction of the Search Spaces we compare

- *p2reach-sdl* with E_{fwd} and E_{bwd} obtained using the priority function defined in Section 4.1.1,
- *p2reach-sdlb* with E_{fwd} and E_{bwd} obtained using no priority function and
- *p2reach-dl* without Search Spaces.

	positive		negative	
	p2reach_	p2reach_s	p2reach_	p2reach_s
	query on large random			
random10m10x	$> 10^7$	**118 587.00**	$> 10^7$	**65 685.80**
random10m2x	205.01	**127.91**	224.83	**132.52**
random10m5x	42 446.50	**1 618.45**	31 001.90	**1 009.51**
random1m10x	1 876 450.00	**40 334.90**	1 153 620.00	**37 504.40**
random1m2x	132.24	**72.55**	142.98	**76.07**
random1m5x	29 463.80	**1 177.91**	20 001.10	**716.34**
	query on large real			
cit-Patents	38 188.20	**1 770.02**	13 355.50	**776.24**
citeseer	10 190.80	**19.14**	**18.06**	21.64
citeseerx	373 587.00	**135.32**	171 640.00	**146.09**
go-uniprot	1 901 810.00	**71.76**	189.22	**109.22**
uniprotenc-100m	3 589 980.00	**60.19**	**61.22**	85.93
uniprotenc-150m	6 879 470.00	**82.10**	**110.05**	116.41
uniprotenc-22m	347 632.00	**16.20**	**16.08**	32.64
	query on small real dense			
arxiv	3 481.36	**659.05**	2 960.09	**2 030.30**
citeseer-sub	284.85	**29.54**	125.91	**50.76**
go	**10.35**	10.89	44.45	**34.59**
pubmed	497.40	**47.69**	163.82	**67.06**
yago	472.03	**8.19**	19.56	**19.07**
	query on small real sparse			
agrocyc	111.19	**7.72**	23.79	**7.13**
amaze	228.33	**5.20**	539.23	**5.27**
anthra	93.99	**7.50**	22.70	**6.86**
ecoo	120.25	**8.93**	26.80	**7.35**
human	110.57	**8.36**	19.91	**7.36**
kegg	217.18	**6.24**	620.97	**6.14**
mtbrv	110.21	**8.38**	26.36	**7.26**
nasa	28.05	**9.34**	57.78	**13.30**
vchocyc	98.78	**8.54**	24.77	**7.44**
xmark	73.23	**37.75**	141.05	**17.83**
	query on stanford			
email-EuAll	24 598.30	**15.00**	4 786.67	**19.34**
p2p-Gnutella31	26.73	**7.04**	716.31	**8.28**
soc-LiveJournal1	68 717.30	**25.76**	175 111.00	**25.53**
web-Google	23 431.80	**23.68**	55 371.00	**25.14**
wiki-Talk	2 713.15	**25.98**	28 303.80	**32.65**

Table 5.3: Query times of 100 000 positive or negative queries in milliseconds. Comparing P2REACH with and without Search Spaces (both without DFS-Trees and without Toplogical Levels)

Other than in Section 5.2.1, we enable our pruning and shortcutting techniques for all configurations we use in this Section.

We see in Table 5.4 and Table 5.5 that *p2reach-sdlb* and *p2reach-dl* have nearly the same query times on all graphs. Whereas *p2reach-sdl* provides a great speed up for dense graphs, on both, positive and negative queries. For positive queries on `random10m10x`, *p2reach-sdl* is nearly 18 times faster than *p2reach-sdlb* and *p2reach-dl*. Therefore, Search Spaces obtained using our priority function enable P2REACH to scale on large and dense graphs.

5.2.3 Topological Levels and DFS-Trees

P2REACH uses Topological Levels and DFS-Trees to prune the search space during the BFS, further we use DFS-Trees to shortcut if possible. We compare

- *p2reach-sdl* with all methods enabled,
- *p2reach-sl*, which is missing DFS-Trees,
- *p2reach-sd*, which is missing Topological Levels and
- *p2reach-s*, which is missing both

on positive and negative queries.

We see in Table 5.6 that for positive queries DFS-Trees provide a speedup up to a factor of 15 on `arxiv`. Best query times for positive queries come from either *p2reach-sdl* or *p2reach-sd*. The latter performs better in most cases due to less overhead of negative pruning checks. Looking at the query times of *p2reach-sl* we can see that checking for Topological Levels slows the BFS down in most cases, but as we can see for `rand10m10x` and `rand1m10x` it can even provide a minor speedup on large dense graphs.

In Table 5.7 we see that *p2reach-sdl* performs the best on nearly all graphs for negative queries. If we compare *p2reach-sl* and *p2reach-sd* with *p2reach-s* we observe that both pruning methods provide a good speedup. Combined we achieve a speedup up to a factor of nearly 100 for `arxiv`.

5.2.4 Query Time Distribution

Figures 5.1, 5.2 and 5.3 show the query time distribution of 100000 random queries on our test graphs. We used the `<chrono>` classes of `C++11` to obtain query times in nanoseconds. We chose box-and-whisker plots to summarize the distribution using whiskers, which represent the lowest time still within 1.5 times the inner quartile range of the lower quartile, and the highest time still within 1.5 times the inner quartile range of the upper quartile. The red flier points are representing times that extend beyond the whiskers, called outliers.

Throughout all plots we see, that the lower limit for a query is around 100 nanoseconds. For small graphs, the median of the query time is between 100 and 120 nanoseconds. Even the third quartile ranges around 100 to 120 nanoseconds for most small graphs. Only `pubmed, arxiv` and `citeseer-sub` have a third quartile of up to 300. Concerning small real sparse graphs, the outliers mainly stay well below 1 000 nanoseconds. For small real dense graphs, the outliers extend up to 40 microseconds. In Figure 5.2, we observe that the query times on graphs of the stanford dataset have slightly higher medians, still below 200 nanoseconds. The according outliers rarely extend above 1 000 nanoseconds.

Figure 5.3 depicts the query distributions on the large real and large random test sets. For large real graphs, the query time distribution for negative is similar to previous test sets. The medians range between 200 and 300 nanoseconds, and the third quartile stays below 300 nanoseconds. The whiskers reach out up to 500 nanoseconds. For `cit-Patents` the outliers

	positive		
	p2reach_sdl	p2reach_sdlb	p2reach_dl
	query on large random		
random10m10x	**78 060.90**	1 281 320.00	1 384 970.00
random10m5x	**1 457.01**	6 628.25	7 567.84
random10m2x	**71.47**	76.02	78.77
random1m10x	**19 982.90**	134 655.00	143 967.00
random1m5x	**1 009.24**	4 435.46	5 038.19
random1m2x	**30.41**	33.74	34.75
	query on large real		
cit-Patents	**1 438.80**	7 828.89	9 221.59
citeseer	**4.00**	4.02	4.00
citeseerx	**49.20**	113.98	119.56
go-uniprot	**45.23**	528 002.00	529 803.00
uniprotenc-100m	5.46	**5.37**	5.39
uniprotenc-150m	6.08	**6.01**	6.01
uniprotenc-22m	**3.60**	3.60	3.61
	query on small real dense		
arxiv	**40.52**	86.12	93.31
citeseer-sub	**12.78**	25.97	30.68
go	5.05	5.10	**5.04**
pubmed	**20.17**	66.44	81.31
yago	**4.39**	8.39	11.80
	query on small real sparse		
agrocyc	1.26	1.21	**1.21**
amaze	1.25	**1.22**	1.23
anthra	1.25	1.23	**1.22**
ecoo	1.29	1.23	**1.22**
human	1.31	1.26	**1.26**
kegg	1.35	1.37	**1.34**
mtbrv	1.25	1.22	**1.21**
nasa	2.25	2.32	**2.21**
vchocyc	1.26	**1.23**	1.28
xmark	**4.15**	5.69	5.97
	query on stanford		
email-EuAll	**3.06**	3.26	3.21
p2p-Gnutella31	3.50	3.51	**3.46**
soc-LiveJournal1	**5.02**	5.32	5.32
web-Google	**5.09**	5.38	5.46
wiki-Talk	6.12	6.34	**6.10**

Table 5.4: Query times of 100 000 positive queries in milliseconds. Comparing P2REACH with Search Spaces obtained using a priorty queue, without using a priorty queue and without Search Spaces. All pruning and shortcutting techniques are enabled.

	negative		
	p2reach_sdl	p2reach_sdlb	p2reach_dl
	query on large random		
random10m10x	**14 561.60**	122 477.00	131 245.00
random10m5x	**497.67**	1 355.04	1 495.10
random10m2x	58.91	56.75	**55.17**
random1m10x	**4 250.29**	13 560.40	14 306.60
random1m5x	**324.51**	885.78	959.64
random1m2x	21.46	20.64	**19.91**
	query on large real		
cit-Patents	**154.46**	361.12	397.51
citeseer	**3.49**	3.52	3.60
citeseerx	**13.12**	16.93	16.82
go-uniprot	**5.47**	22.92	23.13
uniprotenc-100m	**7.72**	7.85	7.89
uniprotenc-150m	**10.36**	10.38	10.43
uniprotenc-22m	3.24	**3.22**	3.23
	query on small real dense		
arxiv	17.85	**16.70**	17.19
citeseer-sub	**6.11**	8.29	8.73
go	3.07	3.19	**3.05**
pubmed	**5.09**	5.66	5.80
yago	**1.96**	3.66	4.12
	query on small real sparse		
agrocyc	0.72	0.70	**0.70**
amaze	0.87	**0.85**	0.87
anthra	0.69	**0.68**	0.68
ecoo	0.71	**0.70**	0.71
human	0.75	**0.74**	0.74
kegg	0.92	**0.89**	0.90
mtbrv	0.72	**0.70**	0.75
nasa	2.13	2.13	**1.89**
vchocyc	0.73	**0.71**	0.76
xmark	2.21	2.08	**1.97**
	query on stanford		
email-EuAll	**2.82**	2.87	2.86
p2p-Gnutella31	1.00	**0.99**	1.04
soc-LiveJournal1	2.96	**2.94**	3.02
web-Google	**4.23**	4.24	4.26
wiki-Talk	3.48	**3.45**	3.96

Table 5.5: Query times of 100 000 negative queries in milliseconds. Comparing P2REACH with Search Spaces obtained using a priorty queue, without using a priorty queue and without Search Spaces. All pruning and shortcutting techniques are enabled.

	positive			
	p2reach_sdl	p2reach_s	p2reach_sd	p2reach_sl
	query on large random			
random10m10x	**78 060.90**	119 507.00	87 189.80	83 091.20
random10m5x	**1 457.01**	1 609.27	1 460.53	1 647.99
random10m2x	71.47	122.33	**70.66**	128.53
random1m10x	**19 982.90**	40 567.70	23 307.80	23 230.30
random1m5x	**1 009.24**	1 183.01	1 149.51	1 182.51
random1m2x	30.41	68.64	**29.57**	74.66
	query on large real			
cit-Patents	**1 438.80**	1 780.24	1 456.08	1 942.98
citeseer	4.00	17.45	**3.46**	19.36
citeseerx	49.20	130.15	**48.31**	147.50
go-uniprot	45.23	68.98	**44.69**	74.58
uniprotenc-100m	5.46	58.53	**4.48**	61.16
uniprotenc-150m	6.08	82.71	**5.10**	86.47
uniprotenc-22m	3.60	15.05	**2.81**	15.89
	query on small real dense			
arxiv	40.52	648.04	**40.15**	536.14
citeseer-sub	12.78	29.06	**12.42**	32.30
go	5.05	10.09	**4.84**	11.26
pubmed	**20.17**	47.04	20.51	46.00
yago	4.39	7.58	**4.04**	9.30
	query on small real sparse			
agrocyc	1.26	7.45	**1.08**	8.55
amaze	1.25	4.72	**1.01**	5.27
anthra	1.25	7.15	**1.08**	8.08
ecoo	1.29	8.61	**1.12**	8.95
human	1.31	7.97	**1.13**	8.53
kegg	1.35	5.50	**1.10**	6.16
mtbrv	1.25	8.33	**1.07**	9.40
nasa	2.25	9.01	**2.00**	9.85
vchocyc	1.26	8.20	**1.08**	8.46
xmark	4.15	35.82	**3.76**	34.63
	query on stanford			
email-EuAll	3.06	12.20	**2.57**	13.91
p2p-Gnutella31	3.50	6.28	**3.24**	7.12
soc-LiveJournal1	5.02	25.00	**4.47**	27.75
web-Google	5.09	21.21	**4.54**	23.97
wiki-Talk	6.12	22.96	**5.65**	25.11

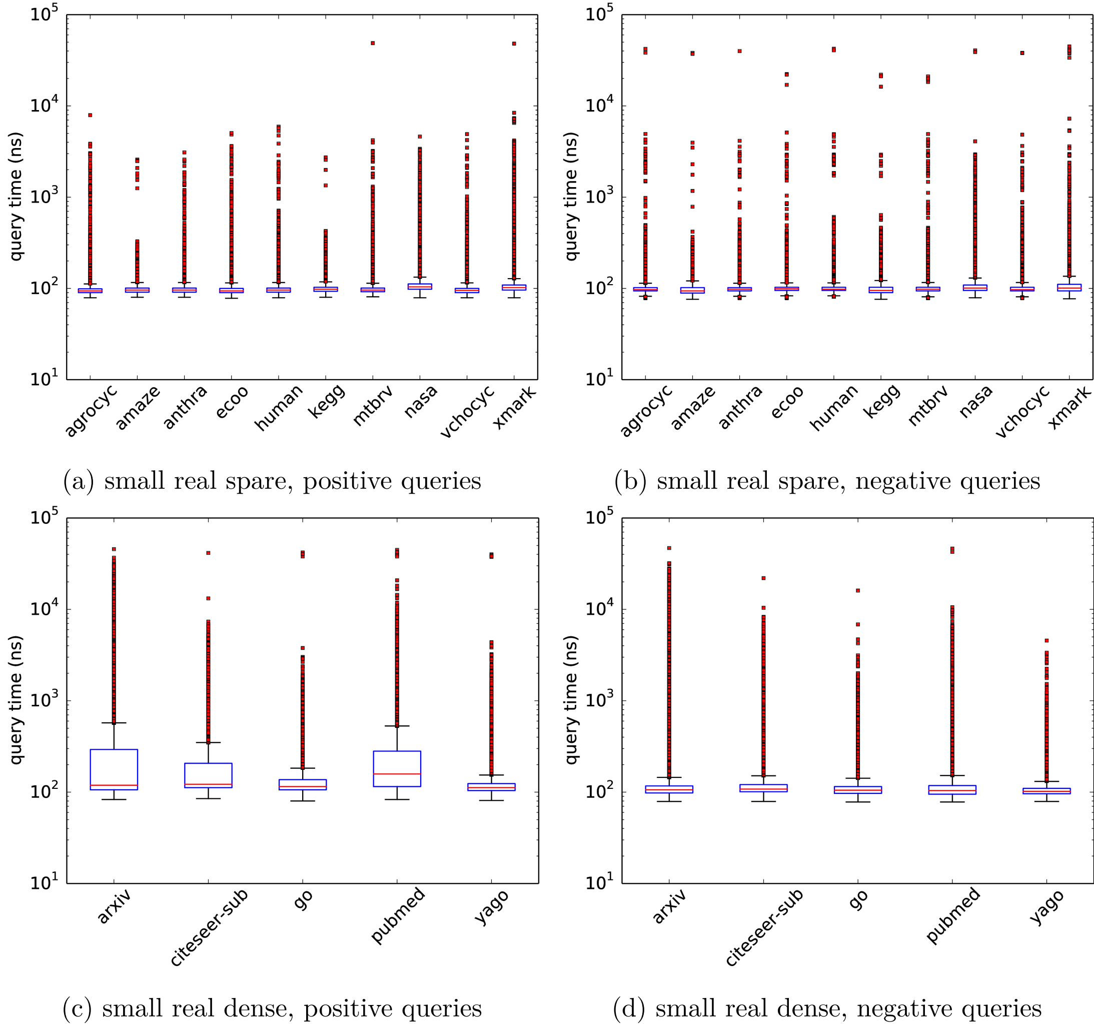
Table 5.6: Query times of 100 000 positive queries in milliseconds. Comparing P2REACH with different combinations of Topological Levels and DFS-Trees

	negative			
	p2reach_sdl	p2reach_s	p2reach_sd	p2reach_sl
	query on large random			
random10m10x	**14 561.60**	65 684.10	18 850.30	15 519.60
random10m2x	**58.91**	128.42	73.31	71.92
random10m5x	**497.67**	995.20	580.06	501.12
random1m10x	**4 250.29**	37 722.00	6 396.85	4 969.61
random1m2x	**21.46**	73.79	32.91	30.46
random1m5x	**324.51**	702.22	387.20	337.10
	query on large real			
cit-Patents	**154.46**	779.37	194.03	207.68
citeseer	**3.49**	19.86	9.53	4.22
citeseerx	**13.12**	143.88	36.89	26.98
go-uniprot	5.47	104.99	34.76	**3.71**
uniprotenc-100m	**7.72**	81.57	53.15	8.73
uniprotenc-150m	**10.36**	114.20	61.22	13.72
uniprotenc-22m	3.24	30.72	17.49	**2.06**
	query on small real dense			
arxiv	**17.85**	1 992.28	34.95	28.84
citeseer-sub	**6.11**	50.73	11.01	10.41
go	**3.07**	32.61	6.96	4.05
pubmed	**5.09**	67.22	9.42	7.45
yago	**1.96**	18.81	5.44	2.51
	query on small real sparse			
agrocyc	**0.72**	6.86	3.29	0.75
amaze	**0.87**	4.64	3.46	0.90
anthra	**0.69**	6.45	3.27	0.69
ecoo	**0.71**	6.86	3.48	0.73
human	0.75	6.22	4.06	**0.73**
kegg	**0.92**	5.39	3.70	1.25
mtbrv	**0.72**	6.86	3.22	0.74
nasa	**2.13**	12.80	4.99	2.56
vchocyc	**0.73**	6.88	3.27	0.81
xmark	**2.21**	16.39	4.60	2.87
	query on stanford			
email-EuAll	**2.82**	16.14	11.41	3.48
p2p-Gnutella31	1.00	7.07	4.04	**0.81**
soc-LiveJournal1	**2.96**	23.89	17.42	5.34
web-Google	**4.23**	23.07	13.29	5.94
wiki-Talk	3.48	29.57	15.33	**3.03**

Table 5.7: Query times of 100 000 negative queries in milliseconds. Comparing P2REACH with different combinations of Topological Levels and DFS-Trees

extend to over one millisecond for one query and for `citeseerx` up to 300 microseconds. Whereas, concerning the rest of the test set, outliers stay below 100 microseconds. For positive queries, we observe a similar result, except for `cit-Patents`, on which the median is 1 000 nanoseconds and the third quartile is at 6 000 nanoseconds.

On large random graphs, the ranges between the first and third quartile grow larger as the graphs get denser. For negative queries the medians stay below 300 nanoseconds, even though the third quartile can range up to 100 microseconds on `random10m10x`. The outliers reach out to more than 10 milliseconds, for the latter. For positive queries the medians and the first quartiles reach higher, up to 200 milliseconds on `random10m10x`. However, the outliers are nearly the same than for negative queries.



(a) small real spare, positive queries

(b) small real spare, negative queries

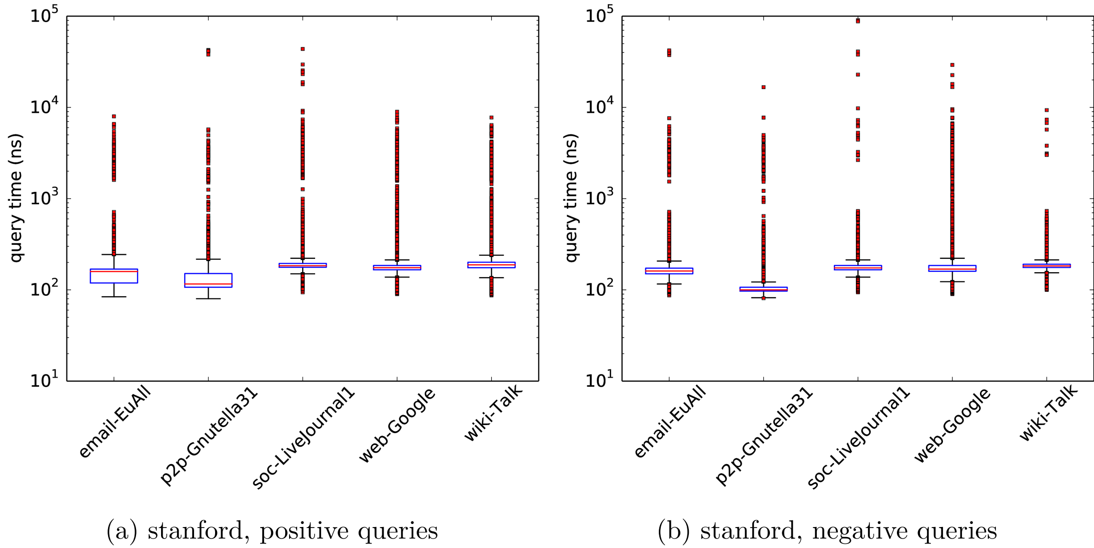
(c) small real dense, positive queries

(d) small real dense, negative queries

Figure 5.1: Query time distribution for positive and negative queries on small real graphs

5.3 Comparison with TF and GRAIL

In this section we compare P2REACH with GRAIL [34], and TF [6]. All algorithms have been publicly provided by the authors and have only been modified to unify their output for automated comparison. Due to memory restrictions some algorithms have been unable to process some input graphs, see the according sections for further information. We used GRAIL with both, two (GRAIL) and five (GRAIL5) traversals, which define the number of intervals in the labels. In Section 5.3 we focus on GRAIL5, as it scales better on large graphs. Furthermore, we recommend [6] for a further comparison with other reachability algorithm, as their experiments can be easily converted and compared with our results.



(a) stanford, positive queries (b) stanford, negative queries

Figure 5.2: Query time distribution for positive and negative queries on our stanford graph set

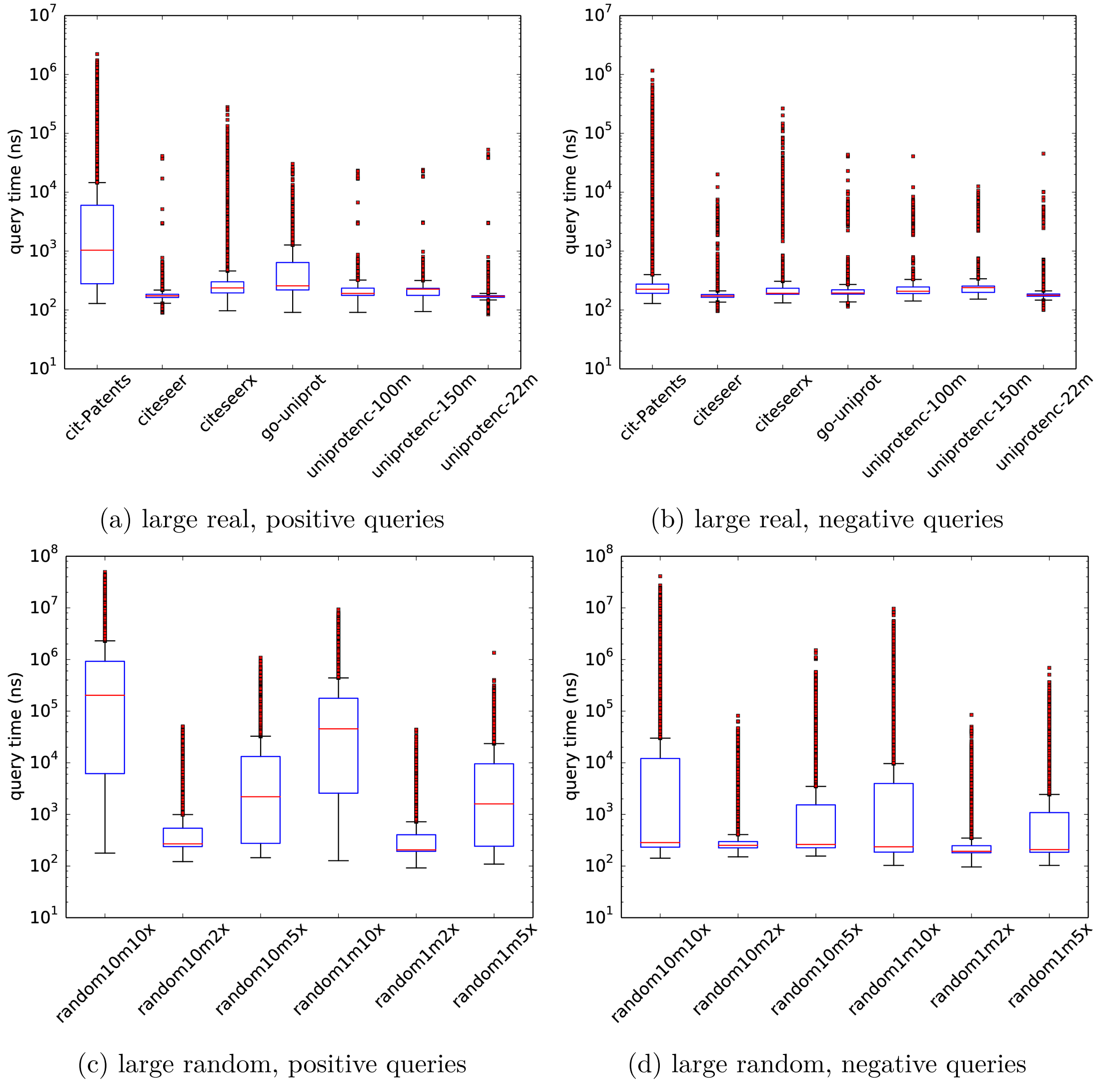
5.3.1 Varying Degree and Size

We used Algorithm 5.1 and a Kronecker graph [21] generator according to the Graph500 benchmark [23] to generate graphs with varying node and edge sets. We directed the edges of the Kronecker graphs by using their integer labels as a topological ordering. In order to compare the impact of the average node degree we generated random DAGs using Algorithm 5.1 with 10,000, 100,000 and 1 million nodes and a varying average node degree from 2^0 to 2^6. Further we generated Kronecker graphs with 2^{16} nodes and varying average node degree between 2^0 and 2^6. To compare the impact of $|V|$ we generated random DAGs using Algorithm 5.1 with varying sizes from 10^4 to 10^8 and $|E|/|V| = 2$. Also we generated Kronecker graphs with sizes from 2^{10} to 2^{23} with the parameters used in the Graph500 benchmark.

Varying Degree

In Figure 5.4 we see construction and query time for Kronecker graphs with varying degree. P2REACH needs less construction time than GRAIL5 and is on a par with GRAIL, whereas TF is several orders of magnitude slower and was unable to handle degrees beyond 2 on 48GB main memory. Concerning the query times, P2REACH manages to answer the 100000 random queries faster than GRAIL, GRAIL5 and TF. As the graphs get denser P2REACH is more than an order of magnitude faster than GRAIL. For node degrees of 1 and 2 TF provides query times faster than GRAIL but much slower than P2REACH.

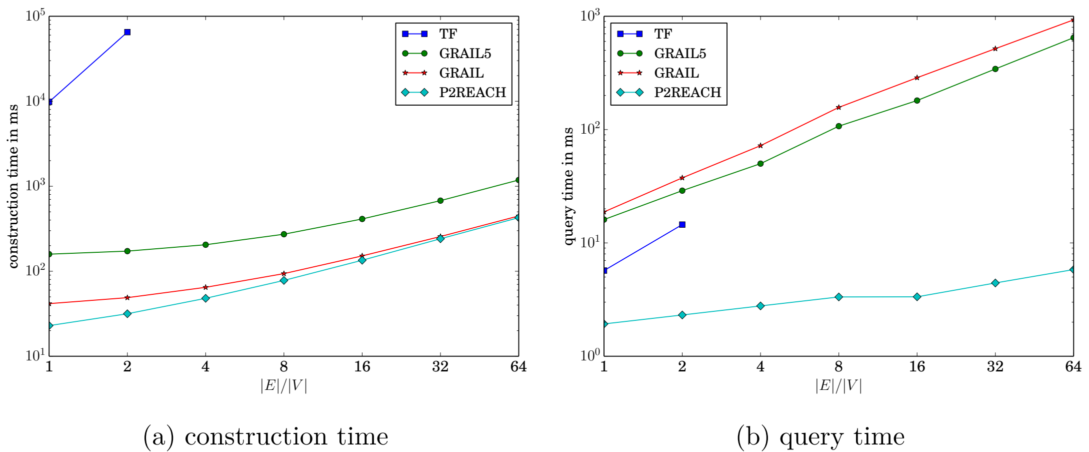
As wee see in Figure 5.5 on random graphs with 10,000 nodes and varying node degree GRAIL and P2REACH have practically the same construction time, GRAIL5 is by a factor of three slower. Beginning with $|E|/|V| = 8$ TF is several orders of magnitude slower in construction than P2REACH and finally failed on a node degree of 64 due to memory consumption. For query processing TF and P2REACH are very similar, P2REACH is three time faster on $|E|/|V| = 1$ whereas TF is three times faster than P2REACH on $|E|/|V| = 4$. At $|E|/|V| = 32$ P2REACH is faster than TF which fails on the last graph. GRAIL and GRAIL5 are slower by an order of magnitude than P2REACH for denser random graphs regarding query time.



(a) large real, positive queries (b) large real, negative queries

(c) large random, positive queries (d) large random, negative queries

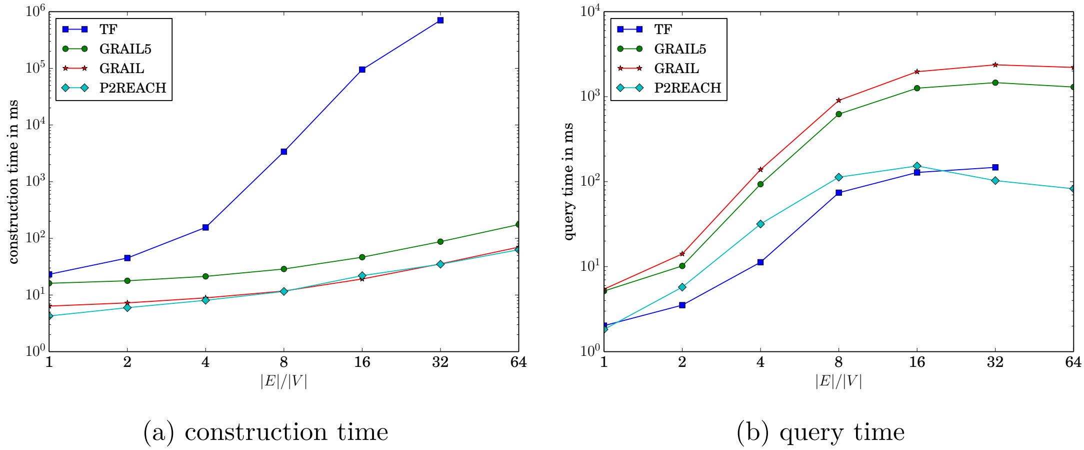
Figure 5.3: Query time distribution for positive and negative queries on large graphs



(a) construction time (b) query time

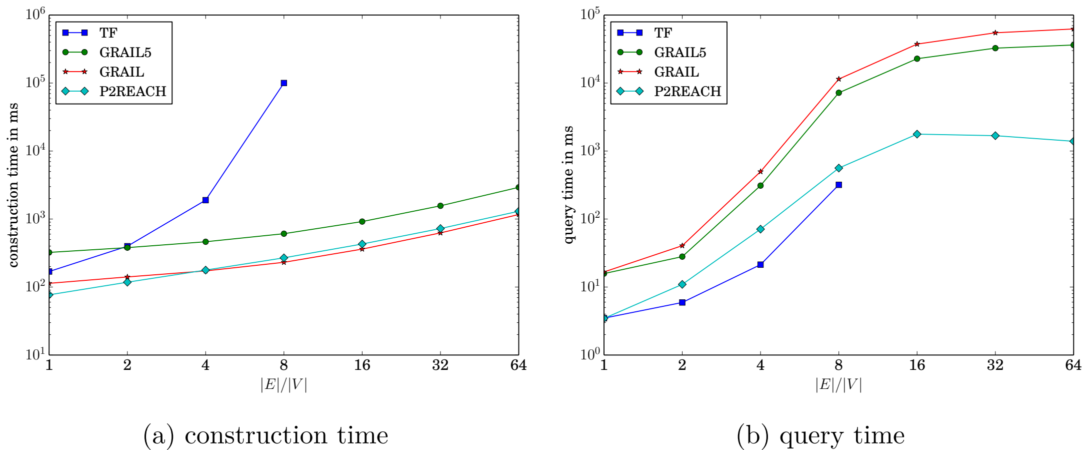
Figure 5.4: Kronecker graphs $|V| = 2^{16}$, varying $|E|/|V|$ from 2^0 to 2^6

In Figure 5.6 we can see that for random graphs with 100,000 nodes and varying degree we get the same result as for random graphs with 10,000 nodes, except TF fails in construction starting at $|E|/|V| = 16$.



(a) construction time (b) query time

Figure 5.5: random DAGs $|V| = 10000$, varying $|E|/|V|$ from 2^0 to 2^6



(a) construction time (b) query time

Figure 5.6: random DAGs $|V| = 100000$, varying $|E|/|V|$ from 2^0 to 2^6

Varying Size

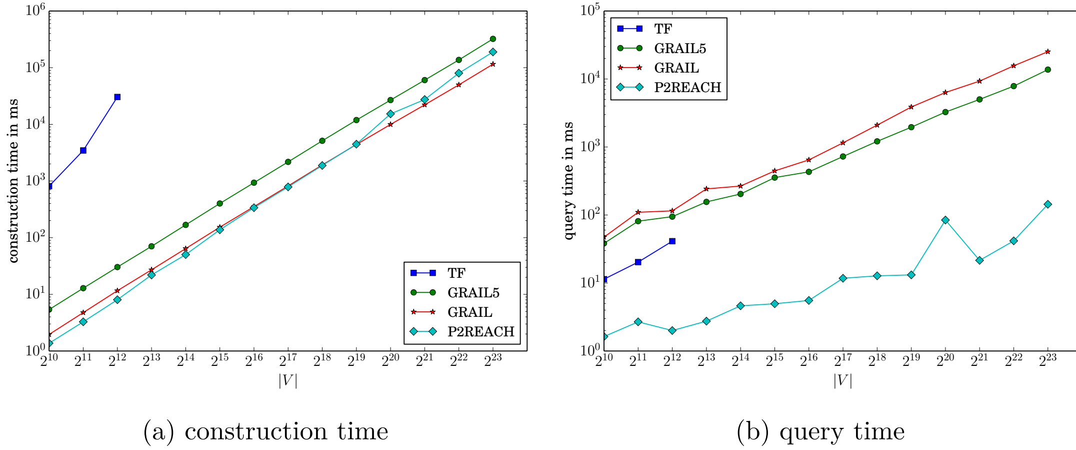
To study the impact of the size of V we generated random graphs with $|V|$ from 10^4 to 10^8 and $|E|/|V| = 2$, random graphs with $|V|$ from 10^4 to 10^7 with $|E|/|V| = 8$ and Kronecker graphs with $|V|$ from 2^{10} to 2^{23} according to the Graph500 benchmark.

Figure 5.7 shows the construction and query times on Kronecker graphs with varying $|V|$. We can observe that GRAIL,GRAIL5 and P2REACH scale linear for the construction time and that P2REACH and GRAIL have nearly the same construction time. P2REACH is slightly faster on small graphs and slightly slower on large graphs, as the $\log n$ factor of the priority queue when constructing the search spaces gets larger than the constant factor

$\lvert V\rvert$	$\lvert E\rvert$	$\lvert V\rvert$	$\lvert E\rvert$
2^{10}	23 735	2^{17}	5 069 222
2^{11}	53 032	2^{18}	10 499 510
2^{12}	117 160	2^{19}	21 631 044
2^{13}	254 721	2^{20}	44 353 834
2^{14}	545 888	2^{21}	90 575 952
2^{15}	1 157 119	2^{22}	184 358 089
2^{16}	2 431 565	2^{23}	374 192 078

Table 5.8: Kronecker graphs with varying $|V|$ and $|E|$

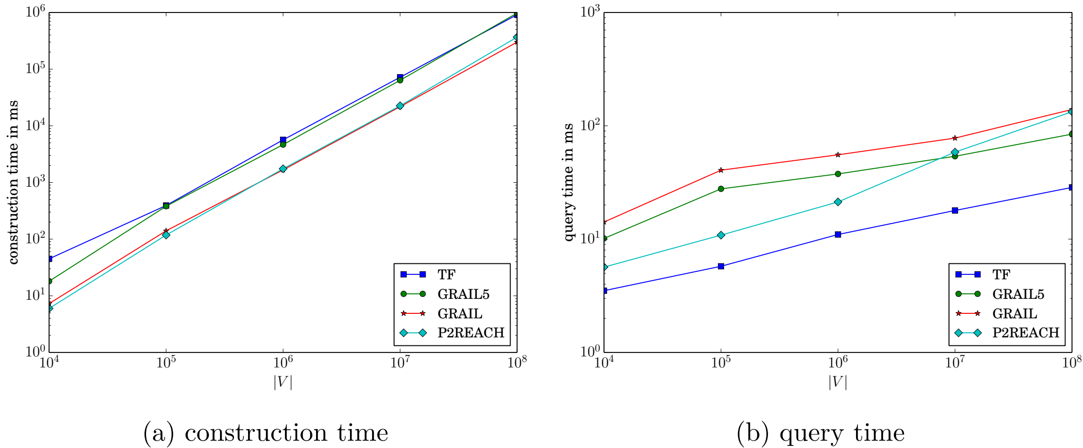
of GRAIL. TF is at least two orders of magnitude slower than P2REACH and fails due to lack of memory on graphs larger than 2^{12} nodes. Regarding query times P2REACH is at least one order of magnitude faster than GRAIL,GRAIL5 and TF.



(a) construction time (b) query time

Figure 5.7: Kronecker graphs with $|V|$ from 2^{10} to 2^{23} and varying $|E|$ see Table 5.8

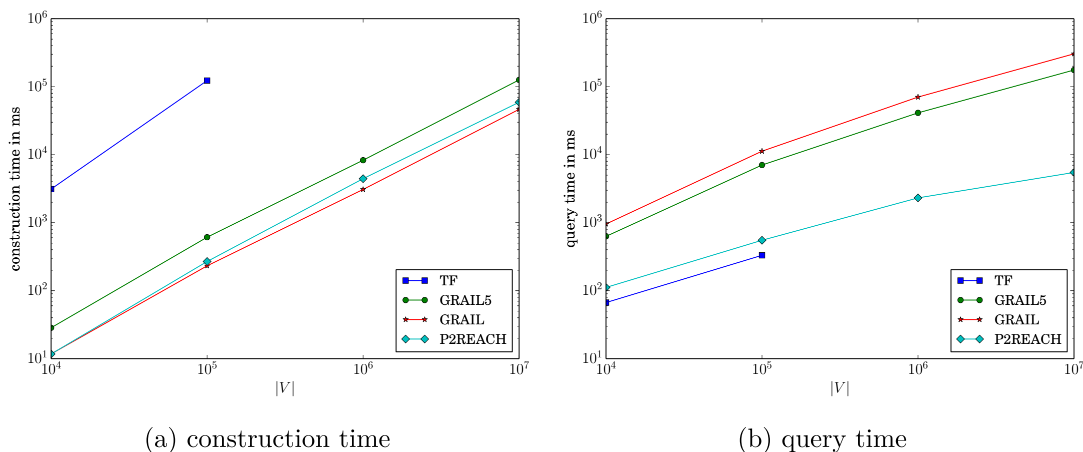
In Figure 5.8 we show construction and query times on random graphs with $|E|/|V| = 2$ with $|V|$ varying from 10^4 to 10^8. All algorithms scale linear regarding node size, where P2REACH and GRAIL are three times faster than TF and GRAIL5. The query times of TF are the best in all cases. As the graphs get larger P2REACH gets slower up to a factor of 4 compared with TF on graphs of size 10^8. Starting at $|V| = 10^7$ GRAIL is slightly faster than P2REACH, as the graphs are thin and even for 100,000 random queries none is positive and GRAIL performs very well on negative queries.



(a) construction time (b) query time

Figure 5.8: Random graphs with $|V|$ from 10^4 to 10^8 and $|E|/|V| = 2$

Furthermore, we study random graphs with $|V|$ between 10^4 and 10^7 and $|E|/|V| = 8$. Similar to Kronecker graphs we can see that the construction times for GRAIL and P2REACH are nearly the same, GRAIL5 is slower by a factor of two. TF has a construction time slower by two orders of magnitude compared with P2REACH and fails to run on graphs larger than 10^5 nodes. Regarding query times P2REACH is an order of magnitude faster than GRAIL and GRAIL5. TF runs slightly faster than P2REACH but as mentioned fails as the graphs grow larger.



(a) construction time

(b) query time

Figure 5.9: Random graphs with $|V|$ from 10^4 to 10^7 and $|E|/|V| = 8$

Index Size

In Figure 5.10 we see the index size of GRAIL, GRAIL5, TF and P2REACH for the previous experiments. The index size accumulates the size of the auxiliary data and in case of GRAIL and P2REACH, the size of $E(G)$. In technical terms, the index size represents the number of integer values an algorithm needs to store.

GRAIL, GRAIL5 and P2REACH have nearly the same index sizes as the have a constant label size per node, and need to store the edge set for traversal. Whereas the index size of TF depends mainly on the density of the graphs and does not scale to dense large graphs. We observed, that the memory consumption of the construction phase of TF scales even worse than the index size.

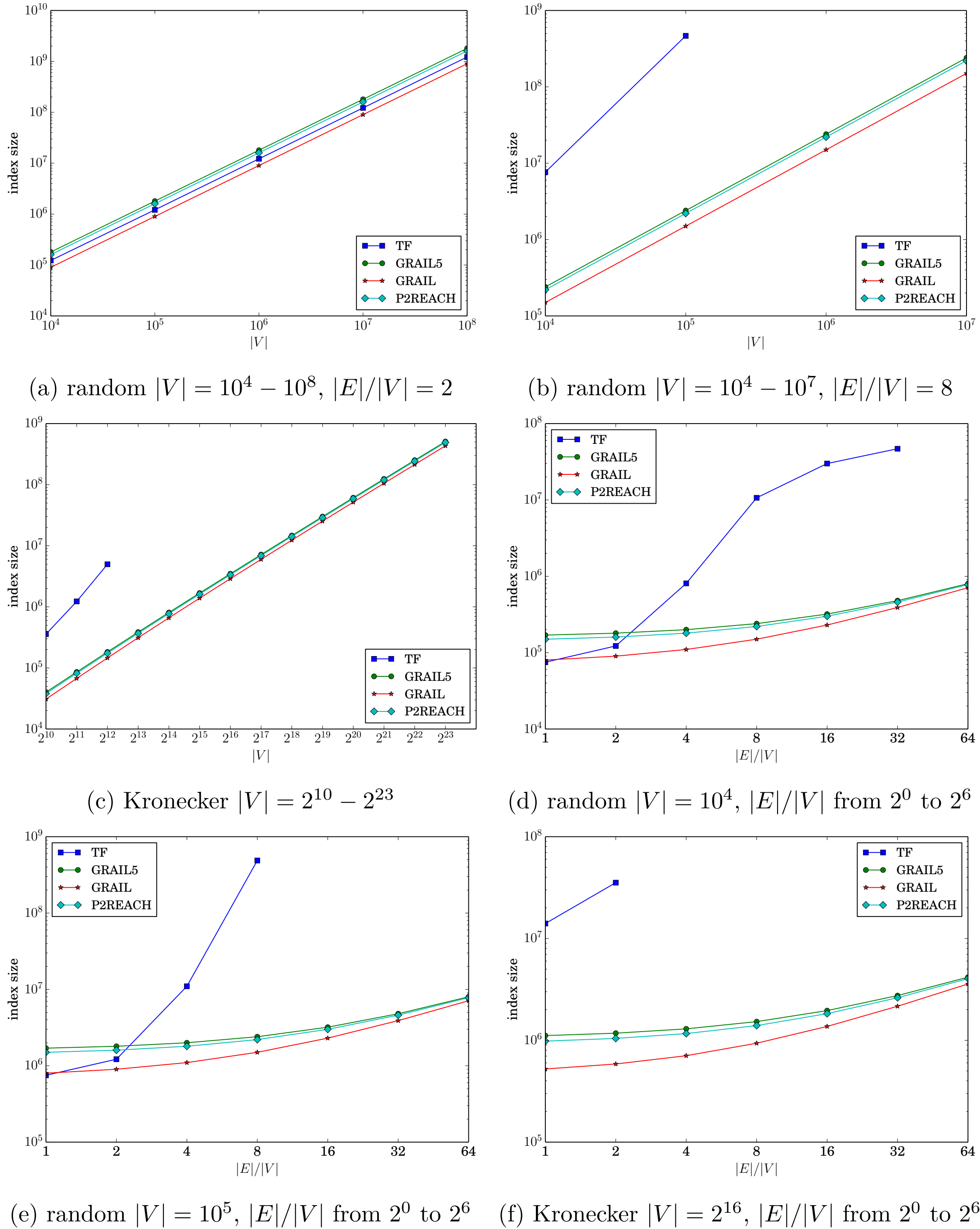
Concerning scaling on degree and size we can finally say that P2REACH scales at least an order of magnitude better than GRAIL and GRAIL5 on large dense graphs. Furthermore, we observe that TF is unable to process those graphs. It performs especially bad on larger Kronecker graphs as they contain long paths. Regarding construction time P2REACH is on a par with GRAIL and scales very good even though it needs a priority queue, which adds an $n \log n$ factor.

5.3.2 Small Real Datasets

Table 5.9 shows the construction time, index size and query time on our small real sparse dataset. As GRAIL and P2REACH need to maintain the edge set of the DAG for traversal and TF does not, we decided to take the edge set into account to calculate the index size needed. Furthermore, we present the index size as ratio to $|V|$, one could say the average label size of a node, despite it also refers to the number of edges in the DAG. We provide the query times for the three of our query types rnd, pos and neg for 100,000 queries respectively. Also we added a column *#POS*, which shows the amount of positive queries for the rnd test sets.

Concerning construction time on the small real sparse dataset, P2REACH performs best and is a least one order of magnitude faster than TF and at least four times faster than GRAIL. As for index size TF provides the smallest labels up to two times smaller than P2REACH.

The query times of P2REACH are at least three times smaller than those of GRAIL and up to two times smaller than those of TF. This holds for random, positive and negative queries and especially on positive queries P2REACH performs an order of magnitude faster than GRAIL.



(a) random $|V| = 10^4 - 10^8$, $|E|/|V| = 2$

(b) random $|V| = 10^4 - 10^7$, $|E|/|V| = 8$

(c) Kronecker $|V| = 2^{10} - 2^{23}$

(d) random $|V| = 10^4$, $|E|/|V|$ from 2^0 to 2^6

(e) random $|V| = 10^5$, $|E|/|V|$ from 2^0 to 2^6

(f) Kronecker $|V| = 2^{16}$, $|E|/|V|$ from 2^0 to 2^6

Figure 5.10: Index sizes, in number of entries, of graphs with varying $|V|$ and varying $|E|/|V|$

The results of our small real dense dataset are displayed in Table 5.10. Again P2REACH performs the best regarding construction time, P2REACH is at least one order of magnitude faster than TF and twice as fast as GRAIL. On `arxiv` the construction time of TF is three orders of magnitude slower than of P2REACH.

The index size of TF on `arxiv` is 30 times larger than of P2REACH. In general the index sizes on the small real dense dataset vary slightly but stay in close range. Query times are in a closer range then on the sparse dataset.

TF performs best for random, negative and positive queries on most graphs of this dataset. P2REACH however, stays in close range for random and negative queries, and is at most a

	GRL5	TF	P2R
agrocyc	15.95	26.54	**2.18**
amaze	5.03	9.54	**0.72**
anthra	15.66	43.46	**2.11**
ecoo	16.60	33.57	**2.15**
human	70.16	75.18	**6.89**
kegg	4.82	9.85	**0.77**
mtbrv	11.74	17.49	**1.66**
nasa	6.59	18.59	**1.41**
vchocyc	11.89	41.41	**1.62**
xmark	7.28	23.60	**1.35**

(a) construction time in ms

	GRL5	TF	P2R
agrocyc	17	**13**	15
amaze	17	**7**	15
anthra	17	**12**	15
ecoo	17	**13**	15
human	17	**9**	15
kegg	17	**7**	15
mtbrv	17	**8**	15
nasa	17	**11**	15
vchocyc	17	**14**	15
xmark	17	**11**	15

(b) index size, in number of integers per node

	random				positive			negative		
	GRL5	TF	P2R	#POS	GRL5	TF	P2R	GRL5	TF	P2R
agrocyc	4.37	1.67	**0.73**	105	23.67	21.54	**1.28**	3.74	1.66	**0.72**
amaze	7.22	1.19	**1.07**	17 075	25.41	1.70	**1.28**	2.80	0.95	**0.88**
anthra	3.82	1.55	**0.69**	98	23.00	16.46	**1.25**	3.98	1.52	**0.69**
ecoo	3.92	1.65	**0.74**	103	23.89	3.83	**1.30**	3.91	1.68	**0.72**
human	6.51	1.44	**0.74**	18	23.81	22.22	**1.34**	6.41	1.43	**0.75**
kegg	8.05	1.35	**1.19**	20 174	26.09	1.95	**1.30**	3.01	1.02	**0.97**
mtbrv	3.66	1.47	**0.73**	157	23.79	1.82	**1.31**	3.60	1.34	**0.72**
nasa	5.59	2.56	**2.05**	552	29.60	4.25	**2.28**	4.79	2.52	**2.07**
vchocyc	3.71	1.60	**0.72**	146	23.45	3.35	**1.26**	3.56	1.64	**0.73**
xmark	6.95	2.61	**2.03**	1 443	24.69	7.69	**4.12**	6.37	2.59	**2.14**

(c) query time for 100 000 queries in ms

Table 5.9: small real sparse graphs

two times slower for positive queries. For random and negative queries P2REACH is even up to two times faster than TF on `arxiv`. Furthermore, P2REACH is two times faster than TF and ten times faster than GRAIL for positive queries on `yago`. Throughout all graphs of this dataset, GRAIL is at least two times slower than TF and P2REACH.

	GRL5	TF	P2R
arxiv	16.77	6 648.44	**6.36**
citeseer-sub	21.92	109.64	**8.18**
go	9.64	33.82	**2.61**
pubmed	17.11	86.86	**7.74**
yago	13.78	47.83	**4.37**

(a) construction time in ms

	GRL5	TF	P2R
arxiv	27	646	**25**
citeseer-sub	20	26	**18**
go	18	**12**	16
pubmed	20	29	**18**
yago	22	**15**	20

(b) index size, in number of integers per node

	random				positive			negative		
	GRL5	TF	P2R	#POS	GRL5	TF	P2R	GRL5	TF	P2R
arxiv	98.50	36.08	**21.43**	15 441	224.43	**26.91**	40.75	54.73	43.16	**18.24**
citeseer-sub	12.71	**4.08**	6.18	379	64.75	**6.86**	12.59	12.34	**4.17**	6.12
go	5.96	**2.92**	3.00	236	27.40	**4.69**	5.00	5.85	**2.94**	3.09
pubmed	14.86	**4.05**	5.20	695	107.25	**9.23**	20.20	13.39	**4.06**	5.14
yago	5.27	**1.70**	1.95	156	43.81	9.35	**4.41**	5.16	**1.67**	1.98

(c) query time for 100 000 queries in ms

Table 5.10: small real dense graphs

5.3.3 Large Synthetic Datasets

In Section 5.2.4 we saw that TF does not scale for dense large graphs, thus TF failed to run on `random1m10x` and `random10m10x` due to memory limitation. In Table 5.11 we see also see that TF performed worst regarding construction on the large random dataset, being more than an order of magnitude slower than P2REACH, which was fastest on all graphs of this dataset. GRAIL was at most by a factor of three slower than P2REACH.

For `random1m2x` and `random10m2x` TF has the smallest index size, slightly smaller than P2REACH followed by GRAIL. Whereas for `random1m5x` and `random10m5x` the index size of TF is one order of magnitude larger than the index sizes of P2REACH and GRAIL.

TF provides the fastest query times for random, positive and negative queries on large random graphs, however only up to a degree of 5, as it failed to run on denser graphs. For $|E|/|V| = 2$ P2REACH and GRAIL are in close range but P2REACH is faster by a factor of four for positive queries. Starting at $|E|/|V| = 5$, P2REACH is at least five times faster than GRAIL and for $|E|/|V| = 10$ P2REACH is over an order of magnitude faster than GRAIL.

	GRL5	TF	P2R
random10m10x	143 619	–	**70 419**
random10m5x	96 466	484 582	**41 892**
random10m2x	63 232	71 630	**22 433**
random1m10x	9 327	–	**5 267**
random1m5x	6 565	42 223	**3 152**
random1m2x	4 618	5 595	**1 751**

(a) construction

	GRL5	TF	P2R
random10m10x	26	–	**24**
random10m5x	21	248	**19**
random10m2x	18	**12**	16
random1m10x	26	–	**24**
random1m5x	21	250	**19**
random1m2x	18	**12**	16

(b) index size, in number of integers per node

	random				positive			negative		
	GRL5	TF	P2R	#POS	GRL5	TF	P2R	GRL5	TF	P2R
random10m10x	589 450	–	**16 611**	5 487	2 002 120	–	**76 301**	262 931	–	**16 620**
random10m5x	2 722	**52**	482	16	7 799	**68**	1 384	2 735	**61**	484
random10m2x	53	**18**	58	0	192	**28**	71	56	**17**	58
random1m10x	84 840	–	**5 179**	9 941	231 149	–	**19 555**	34 005	–	**4 193**
random1m5x	1 835	**44**	327	190	5 253	**55**	975	1 787	**44**	315
random1m2x	37	**11**	21	1	137	**17**	30	37	**11**	21

(c) query time for 100 000 queries in ms

Table 5.11: large random graphs

5.3.4 Large Real and Stanford Datasets

In Table 5.12 we see the results on the large real dataset. P2REACH is at least five times faster than TF regarding construction time except for `citeseer` for which P2REACH still beats TF by a factor of more than two. GRAIL is at least two times slower for the citation networks and for the UniProt RDF graphs it is an order of magnitude slower than P2REACH.

With respect to the index size, TF needs at most half the label size of P2REACH and GRAIL except for `cit-Patents` and `citeseerx`. For `cit-Patents` TF has an index size more 15 times larger than P2REACH and for `citeseerx` its four times larger.

As for the query times, P2REACH provides the smallest query times for random, positive and negative queries on the `uniprotenc` graphs and on `citeseerx`, but TF stays in close range. TF performs best on `cit-Patents`, TF performs the best, beating P2REACH by a factor of three for random and negative queries and by a factor of nearly 20 for positive queries. Furthermore, on `citeseer` TF is about four times faster for negative and random queries than P2REACH, but the latter is twice as fast as TF for positive queries. Throughout all graphs, P2REACH answers negative and random queries between two and five times faster than GRAIL. For negative queries GRAIL is more than an order of magnitude slower, except on `cit-Patents`.

	GRL5	TF	P2R
cit-Patents	22 006	216 741	**9 921**
citeseer	2 213	810	**319**
citeseerx	21 580	84 154	**9 022**
go-uniprot	35 453	63 242	**6 231**
uniprotenc-100m	73 385	39 839	**6 897**
uniprotenc-150m	133 436	56 939	**11 735**
uniprotenc-22m	5 463	2 237	**501**

(a) construction time in ms

	GRL5	TF	P2R
cit-Patents	20	334	**18**
citeseer	16	**6**	14
citeseerx	18	67	**16**
go-uniprot	21	**8**	19
uniprotenc-100m	17	**7**	15
uniprotenc-150m	17	**7**	15
uniprotenc-22m	17	**7**	15

(b) index size, in number of integers per node

	random				positive			negative		
	GRL5	TF	P2R	#POS	GRL5	TF	P2R	GRL5	TF	P2R
cit-Patents	562.48	**53.42**	157.08	52	5 541.42	**85.82**	1 431.63	565.34	**51.58**	150.92
citeseer	11.07	**0.94**	3.46	243	98.41	10.85	**3.98**	11.03	**0.93**	3.53
citeseerx	40.86	19.11	**13.28**	0	756.46	99.62	**48.72**	39.18	18.67	**13.04**
go-uniprot	10.51	**4.87**	5.41	0	153.54	113.46	**44.98**	10.74	**5.13**	5.46
uniprotenc-100m	18.02	9.53	**7.82**	0	115.53	14.21	**5.41**	17.90	9.50	**7.75**
uniprotenc-150m	21.38	11.52	**10.40**	0	136.64	18.09	**5.95**	24.59	11.45	**10.85**
uniprotenc-22m	8.94	3.76	**3.23**	0	82.56	6.67	**3.56**	9.52	3.77	**3.23**

(c) query time for 100 000 queries in ms

Table 5.12: large real graphs

Table 5.13 displays the results of the stanford dataset. P2REACH is nearly twice as fast as TF regarding construction time and 5 to 8 times faster than GRAIL.

TF only needs half the index size of P2REACH and GRAIL for all graphs. Whereas P2REACH provides the fastest query times on all graphs of the test set except on

`email-EuAll` for all query types. P2REACH is nearly two times faster on positive queries than TF and an order of magnitude faster than GRAIL. For random and negative queries TF and P2REACH perform quite alike, whereas GRAIL is around three times slower.

	GRL5	TF	P2R
email-EuAll	655.28	151.55	**78.59**
p2p-Gnutella31	109.30	46.22	**14.83**
soc-LiveJournal1	2 936.28	657.17	**337.45**
web-Google	1 105.72	470.92	**230.89**
wiki-Talk	8 220.39	1 676.31	**987.09**

(a) construction time in ms

	GRL5	TF	P2R
email-EuAll	17	**6**	15
p2p-Gnutella31	17	**7**	15
soc-LiveJournal1	17	**7**	15
web-Google	17	**7**	15
wiki-Talk	17	**7**	15

(b) index size, in number of integers per node

	random				positive			negative		
	GRL5	TF	P2R	#POS	GRL5	TF	P2R	GRL5	TF	P2R
email-EuAll	12.55	**2.15**	2.99	5 163	65.67	9.12	**3.04**	8.90	**1.52**	2.84
p2p-Gnutella31	7.70	1.70	**1.06**	766	38.08	5.40	**3.41**	6.86	1.71	**1.02**
soc-LiveJournal1	25.91	6.12	**3.59**	21 317	75.95	11.14	**5.07**	10.37	3.86	**2.94**
web-Google	25.39	5.63	**4.45**	14 927	73.03	9.30	**5.15**	16.13	4.65	**4.24**
wiki-Talk	10.27	4.24	**3.50**	800	106.11	15.65	**7.33**	11.88	4.08	**3.47**

(c) query time for 100 000 queries in ms

Table 5.13: stanford graphs

6. Conclusion and Future Work

6.1 Conclusion

We engineered an efficient and scalable algorihm to answer reachability queries on directed graphs. P2REACH provides better query performance than other methods that scale comparable and scales better than other methods with comparable query performance. Furthermore, we provide the best positive query performance on a wide range of instances. We introduced several methods for pruning and shortcutting during an online search on a directed acyclic graph, which can be combined with other techniques, and stand out for themselves. In Section 5.3 we saw that TF is unable to process large and dense graphs. Regarding query performance, TF was mostly on a par with P2REACH in our experiments. In the cases TF povided faster query times, it was at the cost of construction time or index size. GRAIL was able to handle all graphs of our experiments but P2REACH provides faster query times, up to an order of magnitude faster, especially for positive queries.

6.2 Future Work

There is still room for improvement left concerning our approach. For once, the auxiliary data used by P2REACH could be compressed and stored more efficiently, since *pmin* and *pmax* are often invalid or useless peek nodes and *ptree* and tl^{till} can be shared between nodes. Furthermore, more work can be done regarding the priority function, that determines the ordering of the node contractions, when constructing the Search Spaces. Similarly, a better heuristic for the ordering of the DFS-Trees could improve the query performance. Additionally, P2REACH could be integrated into the reachability framework introduced by Jin *et al.* [15]. Concerning huge graphs with several billions nodes, a parallel construction phase could be developed and furthermore, optimization for external memory would bring support for even larger graphs. Also, it would be interesting how P2REACH could be modified to handle dynamic cases.

Bibliography

[1] Jørgen Bang-Jensen and Gregory Z Gutin. *Digraphs: Theory, Algorithms and Applications*. Springer, 2009.

[2] Tim Bray, Jean Paoli, C Michael Sperberg-McQueen, Eve Maler, and François Yergeau. Extensible markup language (XML). *World Wide Web Journal*, 2(4):27–66, 1997.

[3] Anita Burgun and Olivier Bodenreider. An ontology of chemical entities helps identify dependence relations among gene ontology terms. In *Proceedings of the First Symposium on Semantic Mining in Biomedicine (SMBM)*, 2005.

[4] Li Chen, Amarnath Gupta, and M Erdem Kurul. Stack-based algorithms for pattern matching on dags. In *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pages 493–504. VLDB Endowment, 2005.

[5] Yangjun Chen and Yibin Chen. An efficient algorithm for answering graph reachability queries. In *Proceedings of the 24th International Conference on Data Engineering (ICDE)*, pages 893–902. IEEE, 2008.

[6] James Cheng, Silu Huang, Huanhuan Wu, and Ada Wai-Chee Fu. TF-Label: A topological-folding labeling scheme for reachability querying in a large graph. 2013.

[7] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and S Yu Philip. Fast computation of reachability labeling for large graphs. In *Proceedings of the 10th International Conference on Extending Database Technology (EDBT)*, pages 961–979. Springer, 2006.

[8] Jiefeng Cheng, Jeffrey Xu Yu, Xuemin Lin, Haixun Wang, and Philip S Yu. Fast computing reachability labelings for large graphs with high compression rate. In *Proceedings of the 11th International Conference on Extending Database Technology (EDBT)*, pages 193–204. ACM, 2008.

[9] Edmund M Clarke. The birth of model checking. In *25 Years of Model Checking*, LNCS, pages 1–26. Springer, 2008.

[10] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. Reachability and distance queries via 2-hop labels. *SIAM Journal on Computing*, 32(5):1338–1355, 2003.

[11] Alin Deutsch, Mary Fernandez, Daniela Florescu, Alon Levy, and Dan Suciu. A query language for XML. *Computer networks*, 31(11):1155–1169, 1999.

[12] Michael L Fredman and Robert Endre Tarjan. Fibonacci heaps and their uses in improved network optimization algorithms. *Journal of the ACM (JACM)*, 34(3):596–615, 1987.

[13] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction hierarchies: Faster and simpler hierarchical routing in road networks. In *Proceedings of the 7th International Conference on Experimental Algorithms*, WEA, pages 319–333, Berlin, Heidelberg, 2008. Springer-Verlag.

[14] H. V. Jagadish. A compression technique to materialize transitive closure. *ACM Trans. Database Syst.*, 15(4):558–598, December 1990.

[15] Ruoming Jin, Ning Ruan, Saikat Dey, and Jeffrey Yu Xu. SCARAB: Scaling reachability computation on large graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 169–180. ACM, 2012.

[16] Ruoming Jin, Ning Ruan, Yang Xiang, and Haixun Wang. Path-tree: An efficient reachability indexing scheme for large directed graphs. *ACM Transactions on Database Systems (TODS)*, 36(1):7, 2011.

[17] Ruoming Jin, Yang Xiang, Ning Ruan, and David Fuhry. 3-hop: A high-compression indexing scheme for reachability query. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, pages 813–826. ACM, 2009.

[18] Ruoming Jin, Yang Xiang, Ning Ruan, and Haixun Wang. Efficiently answering reachability queries on very large directed graphs. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pages 595–608. ACM, 2008.

[19] Markus Krummenacker, Suzanne Paley, Lukas Mueller, Thomas Yan, and Peter D Karp. Querying and computing with BioCyc databases. *Bioinformatics*, 21(16):3454–3455, 2005.

[20] Ora Lassila and Ralph R Swick. Resource description framework (RDF) model and syntax specification. Technical report, W3C, 1999.

[21] Jure Leskovec, Deepayan Chakrabarti, Jon Kleinberg, Christos Faloutsos, and Zoubin Ghahramani. Kronecker graphs: An approach to modeling networks. *The Journal of Machine Learning Research*, 11:985–1042, 2010.

[22] Kurt Mehlhorn and Peter Sanders. *Algorithms and Data Structures: The Basic Toolbox*. Springer, 2008.

[23] Richard C Murphy, Kyle B Wheeler, Brian W Barrett, and James A Ang. Introducing the graph 500. *Cray User's Group (CUG)*, 2010.

[24] Esko Nuutila. An efficient transitive closure algorithm for cyclic digraphs. *Information Processing Letters*, 52(4):207–213, 1994.

[25] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40(11):701–726, 1998.

[26] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61. ACM, 1995.

[27] Robert Rönngren and Rassul Ayani. A comparative study of parallel and sequential priority queue algorithms. *ACM Transactions on Modeling and Computer Simulation (TOMACS)*, 7(2):157–209, April 1997.

[28] Peter Sanders. Fast priority queues for cached memory. *Journal of Experimental Algorithmics (JEA)*, 5:7, 2000.

[29] Ralf Schenkel, Anja Theobald, and Gerhard Weikum. Hopi: An efficient connection index for complex xml document collections. In *Proceedings of the 9th International Conference on Extending Database Technology (EDBT)*, pages 237–255, 2004.

[30] Silke Trißl and Ulf Leser. Fast and practical indexing and querying of very large graphs. In *Proceedings of the 2007 ACM SIGMOD International Conference on Management of Data*, pages 845–856. ACM, 2007.

[31] Sebastiaan J van Schaik and Oege de Moor. A memory efficient reachability data structure through bit vector compression. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data*, pages 913–924. ACM, 2011.

[32] Haixun Wang, Hao He, Jun Yang, Philip S Yu, and Jeffrey Xu Yu. Dual labeling: Answering graph reachability queries in constant time. In *Proceedings of the 22nd International Conference on Data Engineering (ICDE)*, pages 75–75. IEEE, 2006.

[33] Ioannis Xenarios, Danny W. Rice, Lukasz Salwinski, Marisa K. Baron, Edward M. Marcotte, and David Eisenberg. DIP: the database of interacting proteins. *Nucleic Acids Research*, 28(1):289–291, 2000.

[34] Hilmi Yildirim, Vineet Chaoji, and Mohammed J Zaki. GRAIL: Scalable reachability index for large graphs. *Proceedings of the VLDB Endowment*, 3(1-2):276–284, 2010.