# Theory and Engineering
for
Shortest Paths and Delay Management

zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften

der Fakultät für Informatik
des Karlsruher Instituts für Technologie (KIT)

genehmigte

Dissertation

von

Reinhard Bauer

aus Feuchtwangen

Tag der mündlichen Prüfung:	08. Dezember 2010
Erster Gutachter:	Frau Prof. Dr. Dorothea Wagner
Zweiter Gutachter:	Frau Prof. Dr. Anita Schöbel

Acknowledgments

First of all, I would like to thank my supervisor Dorothea Wagner for her constant support, the opportunity to work in her great group and for always being available for her employees' concerns. I am deeply grateful to Anita Schöbel for her selfless effort, the delightful working-atmosphere she creates and the willingness to review this thesis.

Many thanks go to my colleagues in Dorothea Wagner's group for providing support as well as distraction and a friendly, stimulating environment. Special thanks go to all the people who helped proofreading this thesis. Very special thanks go to Marcus Krug for uncomplainingly reading Chapter 3 twice. A big thank-you goes to my co-authors for nice discussions and good collaboration.

Besides that, I would like to thank all the people that supported this thesis outside the office: Most of all, Elena, for her neverending support, patience and understanding. Further, the KAP-guys for the distraction from work and Daniel Gentner, Martin Küster and Max Stengel for always having an open ear. Last but not least, I would like to express my deep gratitude to my parents Curt and Gertrud Bauer who supported me in countless ways.

Contents

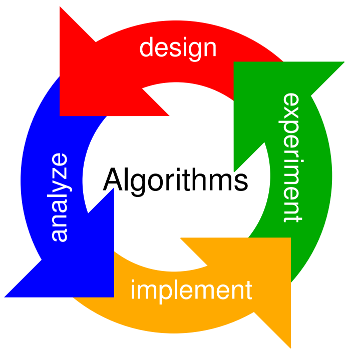
Nomenclature

$\mathbb{N}$	set of non-negative integers	5
$\mathbb{Z}$	set of integers	5
$\mathbb{R}$	set of real numbers	5
$1_A(x)$	indicator function of set A	5
$\mathcal{P}(A)$	power set of set A	5
$\mathrm{dist}(s,t)$	distance from node s to node t	7
$N(v)$	neighborhood of node v	6
$\overleftarrow{G}$	reverse graph of G	6
$\epsilon_G(v)$	eccentricity of node v	7
$V_\alpha(s,t,G)$	search space of an unidirectional speedup technique	18
$V_\alpha^+(s)$	forward search space of a bidir. speedup technique	18
$V_\alpha^-(t)$	backward search space of a bidir. speedup technique	18
$G[S]$	graph G augmented by shortcut assignment S	19
$\mathrm{SP}(G)$	set of all shortest paths in G	20
$\mathcal{R}(v)$	reach of node v	20
$P^-(x,y)$	set of start nodes of shortest paths through x and y	68
$P^+(x,y)$	set of end nodes of shortest paths through x and y	68
$P(x,y)$	set of pairs of nodes s, t, for which there is a a shortest s-t-path containing first x and then y	68
$P^{\bowtie}(x,y)$	set of all nodes that lie on a shortest x-y-path	68
$h_G(s,t)$	hop-distance from s to t	69
$\omega_{sb}(t)$	offset of node t	83
$g_s(a,b)$	gain of a shortcut from a to b with respect to s	83
$\mathrm{spDiam}(G)$	shortest-paths diameter of graph G	87
$\mathrm{con}(v)$	consistent value of node v	94

Chapter 1

Introduction and Outline

Algorithm Engineering is a modern method for algorithm design. The core of this approach is a cycle consisting of the design, theoretical analysis, implementation and experimental evaluation of practicable algorithms. The aim of the experimental evaluation is to gain new insights, guiding both design and theory. Ideally, this leads to an altered point of view on the applied algorithms or even on the underlying problem and revives the algorithm-engineering cycle. On the other hand, theoretical considerations are meant to stimulate the development, improvement and understanding of practically efficient algorithms. Often, a special focus lies on algorithms for real-world data and real-world applications.



Emblem of the DFG Priority Programme 1307 - Algorithm Engineering

Throughout this thesis we are guided by the idea of algorithm engineering. For each considered problem field, a reasonable next step in the spirit of algorithm engineering is determined and carried out. We address problems in the area of shortest-paths computation and algorithms for infrastructure networks. The work can logically be separated into four problem fields which are sketched in the following outline.

Chapter 2 - Fundamentals. This chapter recapitulates concepts and algorithms used later. Fundamental terminology is established. The treated background ranges from basic graph theory over Dijkstra's algorithm and basic complexity theory to mixed-integer linear programs.

Chapter 3 - Preprocessing Speedup-Techniques is Hard. In 1999, Schulz, Wagner and Weihe published the first work on the fast computation of shortest paths between arbitrary pairs of points in a (railway-)network exploiting a preprocessing phase [SWW99]. This paper was followed by a remarkable 'horse-race' for the fastest technique for the computation of point-to-point shortest paths in large networks. The resulting techniques are called *speedup techniques* as they usually speed up Dijkstra's algorithm. Mostly, these approaches are custom-tailored for road networks and are up to 3.000.000 times faster than Dijkstra's algorithm. A special feature of the problem is the availability of large and meaningful real-world data: Graphs representing the road networks of the USA and Europe are available for scientific use. Besides the vast amount of work on the core problem 'computation of point-to-point shortest-paths in static graphs' there are also results on extended problems like multi-modal routing, shortest-paths computation in time-dependent graphs or many-to-many shortest path computation.

The results achieved in this field are widely considered to be a showpiece of algorithm engineering. Up to now, the impressive experimental results in the area are not backed up by a theoretical foundation (which very well is part of the idea of algorithm engineering). Recently some first theoretical insights were made that improved the understanding of the huge speedups that had been achieved [AFGW10].

The Problem. The preprocessing phases of most speedup-techniques leave open some degree of freedom like the choice of how to partition a graph or how to insert additional edges into a network. In practice, this degree of freedom is filled in a heuristical fashion. Thus, for a given speedup technique, the question arises how to fill the according degree of freedom optimally.

Our Contribution. We model all according techniques in a common framework and show NP-hardness for filling the respective degree of freedom optimally. The applied objective function is the expected size of the search space of a shortest-path query. Part of this chapter has been published in [BCK$^+$10a, BCK$^+$10b] and is joint work with Tobias Columbus, Bastian Katz, Marcus Krug and Dorothea Wagner.

Chapter 4 - The Shortcut Problem. In this chapter, we study a graph-augmentation problem arising from an approach used in many speedup techniques. Many of those enhance the graph by inserting *shortcuts*, i.e., additional edges (u, v) such that the length of (u, v) equals the distance from u to v in the original graph.

Chapter 3 treated, amongst others, the question of how to insert shortcuts to minimize the expected size of the search space of the applied technique. In this chapter, we study the even more fundamental question of how to minimize the number of edges on edge-minimal shortest paths. This analyzes an aspect of shortcuts that is independent of the applied speedup-technique.

While the idea for the problem formulation clearly stems from algorithm engineering considerations, our motivation for working on that problem deviates a bit from the algorithm engineering idea: We primarily consider the problem to be a straightforward, beautiful theoretical problem on its own right. Accordingly, we work in a mostly theoretical manner and do not infer from our results on the shortcut problem to speedup-techniques.

The Problem. Given a weighted, directed graph G and a number $c \in \mathbb{Z}^+$, the shortcut problem asks how to insert c shortcuts in G such that the expected number of edges that are contained in an edge-minimal shortest path from a random node s to a random node t is minimal.

Our Contribution: We state two variants of the problem and study their algorithmic complexity. We further develop ILP-based exact approaches and consider a greedy strategy. We show an approximation guarantee of the greedy-strategy on a special graph class and give a fast algorithm for performing a greedy step. Finally, we show how to stochastically evaluate a given set of shortcuts on graphs that are too large to do so exactly. Part of this chapter has been published in [BDDW09, BDD$^+$10] and is joint work with Gianlorenzo D'Angelo, Daniel Delling, Andrea Schumm and Dorothea Wagner.

Chapter 5 - Batch-Dynamic Single-Source Shortest-Paths Algorithms. Real-world applications like routing in road networks, data harvesting in sensor networks or routing in the internet require computing and storing shortest-paths trees. Whenever the underlying graph changes, the given shortest-paths tree (or a distance vector implicitly determining it) has to be updated.

Algorithms that update the tree without a full recomputation from scratch are called *dynamic single-source shortest-path algorithms.* Some of the algorithms known in the literature are only able to cope with the update of one edge at a time, while others can perform *batch updates*, i.e., update the shortest-path information after multiple edges have simultaneously changed their weight.

One can consider edge insertions and deletions as special cases of weight changes: Deletions correspond to weight increments to infinity, while insertions are weight decrements from infinity. An algorithm is called *fully dynamic* if both weight increases and decreases are supported, and *semi-dynamic* if only weight decreases or only increases are supported.

The Problem. We study experimental aspects of fully-dynamic single-source shortest-path algorithms for graphs with positive edge weights. We pay special attention to the batch case. Up to now, the experimental knowledge on the topic is quite sparse. The existing experimental work focuses on very specific datasets and there is no systematic comparison of the different approaches. We are not aware of any experimental evaluation of the batch case. Hence, for batch updates it is not even known if it is useful to process a set of updates as a batch.

Our Contribution: We give an extensive experimental study for the single-edge and for the batch case. We further work on an already existing algorithm. This algorithm has been stated with regard to mainly theoretical considerations. We state and test an efficient implementation of this algorithm as well as combinations with other approaches. Furthermore, we propose a simple method to decide if one should handle updates in a batch or iteratively. Finally, the outcomes of our experiments allow some new insights in both the algorithms and the problem itself. Part of this chapter has been published in [BW09a, BW09b] and is joint work with Dorothea Wagner.

Chapter 6 - Practical Online Algorithms for Delay Management. The *delay management problem* asks how to react to exogenous delays in public railway traffic such that the overall passenger delay is minimized. These *source delays* occur in the operational business of public transit and can make the scheduled timetable infeasible.

The two main aspects treated in the literature are as follows: Firstly, passenger trips often require changing from one train to another. Given a delayed feeder train, a *wait-depart decision* settles the question if a follow-up train should wait in order to enable *changing activities*. Secondly, the limited capacity of the track system complicates the creation of a good disposition timetable. *Headway constraints* model this limited capacity. Every time two trains simultaneously compete for the same part at the train system, it has to be decided which train may go first. Typically, an additional obstacle is the online nature of the problem. Source delays are often unknown in advance and decisions have to be taken without exactly knowing the future.

The Problem. In this chapter we search for practical algorithms for online delay management. Our main focus lies on wait-depart decisions but we also briefly consider headway constraints. We aim at finding algorithms that are simple, robust and of good solution quality.

Our Contribution: We enhance an existing offline model and gain a generic model that is able to cover complex realistic memory-less delay scenarios as well as standard academic delay scenarios that require knowing the past. We further introduce and experimentally evaluate online strategies for delay management. While we also test ILP-approaches, our primary aim are strategies that are practical in the sense that they are simple and robust. Hence, we propose strategies that do not need complete information on the state of the entire system. We compare our results to tight a-posteriori bounds given by an optimal offline solution. Finally, by analyzing the solutions found, we gain interesting new insights into the structure of good delay-management strategies. This chapter is based on joint work with Anita Schöbel.

Chapter 7 - Conclusion. The thesis ends with a conlusion. We summarize the most important lessons learned, point out some interesting open questions and give an outlook.

Chapter 2

Fundamentals

This chapter recapitulates concepts that are needed later. We establish basic terminology and state Dijkstra's algorithm. We further sketch the foundations of complexity theory and approximability and outline mixed-integer linear programming. We assume the reader to be already familiar with basic mathematical concepts and algorithmics. Hence, this chapter is not intended to give an introduction to the beginner but to clarify terminology and notation in case of ambiguity.

Further information on graph theory and algorithmics can be found in [Jun99] and [CLRS01], more on complexity theory in [GJ79, HMU07], more on approximation algorithms in [ACG$^+$02] and more on mixed-integer linear programming in [NW88].

Sets. We denote by $\mathbb{Z}$ and $\mathbb{R}$ the set of integers and reals, respectively. We use the notation $\mathbb{Z}_{\geq 0} := \{x \in \mathbb{Z} \mid x \geq 0\}$, the symbols $\mathbb{Z}_{>0}$, $\mathbb{R}_{\geq 0}$ and $\mathbb{R}_{>0}$ are defined accordingly. The symbol $\mathbb{N}$ is a synonym for $\mathbb{Z}_{\geq 0}$. Given a set A, the *powerset* $\mathcal{P}(A)$ of A is the set of all subsets of A including the empty set and A itself. Let $A \subseteq X$ be a subset of a set X. The *indicator function* of A and X is the function $1_A : X \to \{0, 1\}$ defined as $1_A(x) = 1$ if $x \in A$ and $1_A(x) = 0$ otherwise.

Dependencies. In this work, we often face the situation that, while a mathematical object is depending on many other objects, most of these dependencies are clear from the context. In order to improve readability, we drop these dependencies in the notation in this case. For instance, we use the notation $\operatorname{dist}(s, t)$ for the distance from node s to node t if the choice of the underlying graph G is clear.

2.1 Graphs

Basic Definitions. A *directed graph* G is an ordered pair $G = (V, E)$ consisting of a finite set V and a set E of ordered pairs (u, v) of elements u and v in V. Elements of V are called *nodes* (or *vertices*), elements of E are called *edges* (or *arcs*). Given an edge (u, v) we call u the *source node* and v the *target node* of (u, v). Further, (u, v) is an *incoming edge* of v and an *outgoing edge* of u. Whenever we have a function $f : E \to M$ from the set of edges E to an arbitrary other set M we abbreviate $f((u, v))$ by $f(u, v)$.

An *undirected graph* G is an ordered pair $G = (V, E)$ consisting of a finite set V and a set E of two-element subsets of V. Again, elements of V are called *nodes* (or *vertices*), elements of E are called *edges*.

Most of the following definitions can be done simultaneously for directed and undirected graphs. In that case, we write (u, v) for both, a directed edge (u, v) and an undirected edge $\{u, v\}$. Unless stated otherwise, all graphs occurring in this thesis are directed. Given a graph $G = (V, E)$ we sometimes write $v \in G$ for $v \in V$ and $(u, v) \in G$ for $(u, v) \in E$.

A *weighted (directed/undirected) graph* is an ordered triple (V, E, len) with (V, E) being a (directed/undirected) graph and $\text{len} : E \to \mathbb{R}$ being a mapping. We call $\text{len}(e)$ the *length* (or *weight*) of edge e.

Given nodes u and v, we call u a *neighbor* of v if there is an edge (u, v) or (v, u) in G. Let $G = (V, E)$ be an undirected graph. We denote by $N(v)$ the *neighborhood* of v, i.e., the set of all neighbors of v. Let $G = (V, E)$ be a directed graph. The *in-degree* of a node $v \in V$ in G is the number of edges (in E) with target v, the *out-degree* of v is the number of edges with source v. Let G be a directed or undirected graph. The *degree* of v in G is the number of edges for which one end-vertex is v.

Any graph $G' = (V', E')$ where $V' \subseteq V$ and $E' \subseteq E$ is called a subgraph of $G = (V, E)$.

Finite and Simple Graphs. An edge of the form (u, u) is called a *loop*. We always work on loopless graphs. In some applications also *infinite graphs*, i.e., graphs with infinite node set V, are of interest. Throughout this work, we do not consider *infinite graphs*. Furthermore, we never generalize the set E of edges to be a multiset. Hence, for a (directed or undirected) graph (V, E) and two nodes $v, w \in V$, the edge (v, w) from v to w always is unique if (v, w) exists. All occurring weighted graphs have positive edge lengths, i.e., $\text{len}(e) > 0$ for all $e \in E$.

Reverse Graph. Let $G = (V, E)$ denote a directed (unweighted) graph. We denote by $\overleftarrow{G}$ the *reverse graph* (of G), i.e., the graph $(V, \overleftarrow{E})$ with $\overleftarrow{E} := \{(v, u) \mid (u, v) \in E\}$. In case $G = (V, E, \text{len})$ is weighted, the reverse graph $\overleftarrow{G}$ is $(V, \overleftarrow{E}, \overleftarrow{\text{len}})$ with $\overleftarrow{E}$ being as above and $\overleftarrow{\text{len}}$ being defined by $\overleftarrow{\text{len}}(u, v) := \text{len}(v, u)$ for $(v, u) \in E$.

Walks and Paths. Let $G = (V, E)$ be a (directed or undirected) graph. A *walk* P from x_1 to x_k in G is a finite sequence $(x_1, x_2, \dots, x_k)$ of nodes such that $(x_i, x_{i+1}) \in E$ for $i = 1, \dots, k-1$. Given a walk $P = (x_1, x_2, \dots, x_k)$, we also consider $P = (\{x_1, \dots, x_k\}, \{(x_i, x_{i+1}) \mid i = 1, \dots, k-1\})$ to be a subgraph of G. Now let $G = (V, E, \text{len})$ be additionally weighted. The *length* $\text{len}(P)$ of P is the sum of the lengths of all edges in P, i.e., $\text{len}(P) = \sum_{i=1}^{k-1} \text{len}(x_i, x_{i+1})$. A *path* is a walk that contains each vertex at most once, i.e., $P = (x_1, x_2, \dots, x_k)$ is a path, if and only if, $x_i \neq x_j$ holds for each $1 \leq i \neq j \leq k$.

Connectivity. We say a graph $G = (V, E)$ is *strongly connected*, if there is an s-t-path for any pair of nodes $s, t \in V$. Given a directed (weighted or unweighted) graph $G = (V, E)$, the corresponding undirected (and unweighted) graph $G' = (V, E')$ is given by $E' = \{\{u, v\} \in V \times V \mid \exists (u, v) \in E\}$. We say a directed graph is *(weakly) connected* if the corresponding undirected graph is strongly connected. Note that, for undirected graphs, weak and strong connectivity are equivalent and we hence only use the term *connected* for these graphs. Given nodes s and t, we say t is *reachable* from s (in a graph G') if there is an s-t-path (in G').

Cycles, Trees and Acyclic Graphs. A *cycle* C is a walk starting and ending at the same node that contains each edge at most once and that contains at least two nodes.

A *directed acyclic graph* is a directed graph without cycles. We call a graph $T = (V', E')$ a *tree* if there is a node s such that for each node $t \in V'$ there is exactly one path from s to t and such that T is acyclic. We call s a *root* of T. We say a vertex v is a *descendant* of a vertex t in T (with respect to root s), if the path from s to v in T contains t. Note that each node is a descendant of itself. Given a tree T with root s and a vertex w in T, we call v the *parent* of w (on T with respect to s) if the last edge of the s-w-path in T is (v, w).

A *topological order* $\prec$ of a directed acyclic graph $G = (V, E)$ is a reflexive, antisymmetric, transitive and total relation on V such that for each edge (u, v) in E, $u \prec v$ holds. If G is acyclic, a topological ordering can be computed in time $O(|V| + |E|)$ [Jun99].

Shortest Paths and Distances. Let $G = (V, E, \text{len})$ be a graph and $s, t \in V$ be nodes. A *shortest path* from node s to node t is a path from s to t of minimum length. We call a shortest path from s to t a *shortest s-t-path.* Given two nodes s and t, the distance $\text{dist}(s, t)$ from s to t is the length of a shortest s-t-path and infinity if no s-t-path exists. Note that the distance is well-defined as we only work on finite graphs with non-negative edge lengths.

The *diameter* of G is the largest distance in G, i.e., $\max\{\text{dist}(s, t) \mid s, t \in V\}$. The *eccentricity* $\epsilon(v)$ of a node v is the maximum distance from v to any other node t in G, i.e., $\epsilon(v) = \max\{\text{dist}(v, t) \mid t \in V\}$.

A *shortest-paths tree with root s* is a subgraph $T = (V', E')$ of G such that T is a tree, V' is the set of nodes reachable from s and such that for each edge $(u, v) \in E'$ we have $\text{dist}(s, u) + \text{len}(u, v) = \text{dist}(s, v)$. Note that each path in T is a shortest path in G. The *shortest-paths subgraph with root s* is the subgraph $G_s = (V', E'')$ of G such that V' is the set of nodes reachable from s and E'' is the set of all edges with $\text{dist}(s, u) + \text{len}(u, v) = \text{dist}(s, v)$. Note that the paths in G_s that start with s are exactly the shortest paths in G that start with s. Further, G_s is directed acyclic in case all edge weights are strictly positive.

The *triangle inequality* for graphs states that $\text{dist}(s, u) + \text{dist}(u, t) \geq \text{dist}(s, t)$ for any triple s, u, t of nodes.

2.2 Dijkstra's Algorithm

In the *single-source shortest-paths problem* we are given a graph $G = (V, E, \text{len})$ and a source $s \in V$. The objective is to compute the distance from s to all nodes in V. Sometimes we additionally want to compute a shortest-paths tree or the shortest-paths subgraph with root s. We describe *Dijkstra's algorithm* which solves the single-source shortest-paths problem in graphs with positive edge lengths.

We start by outlining the *Bellman-Ford Equations* which are a main argument for the correctness of the approach. We then specify the datastructure *priority queue* which is one of the main ingredients of Dijkstra's algorithm. This is followed by the description of Dijkstra's algorithm.

Bellman-Ford Equations. The Bellman-Ford Equations describe the single-source shortest-paths problem in a more local fashion. Given the graph $G = (V, E, \text{len})$ and the source s, we have a variable d_v for each node $v \in V$. The equations are

$$\begin{aligned} d_s &= 0 \\ d_v &= \min\{d_u + \text{len}(u, v) \mid (u, v) \in E\} \qquad , v \in V \setminus \{s\} \ . \end{aligned}$$

If all Bellman-Ford Equations are fulfilled d_v equals, for each $v \in V$, the distance from s to v.

Priority Queues. A *priority queue* is a datastructure that maintains a set S of tuples (u, key) where u is an arbitrary object and key is a real value. We call key also the *priority* of u. Note that sometimes generalized variants are applied, for which key is element of an arbitrary totally-ordered set. There are various types of priority queues. Throughout this work, we do not require to know the type of the applied priority queue when working theoretically. However, we always use a *binary heap* when performing experiments (see [CLRS01] for a description).

In the literature, the operations supported by a priority queue differ. The operations we require in this thesis are given in Table 2.1. The operations InsertOrUpdate and Remove can easily be built from the standard priority queue operations. However, for binary heaps, the Remove-operation can be implemented more efficiently than using the standard operations. We include both operations explicitly as we often use them.

Dijkstra's Algorithm. Given a graph $G = (V, E, \text{len})$ with positive length function $\text{len} : E \to \mathbb{R}_{>0}$ and a node $s \in V$, Dijkstra's algorithm [Dij59] finds the distances from s to all nodes in the graph. We state a variant that additionally computes a shortest-paths tree with root s. Accordingly, we call s the *root*. It is straightforward to adapt the algorithm to also compute the shortest-paths subgraph with root s.

For each node v in the graph, the algorithm maintains a distance label $d(v)$. During the run of the algorithm $d(v)$ contains an upper bound for the distance from s to node v, after termination $d(v)$ equals the exact distance $\text{dist}(s, v)$. The shortest-paths tree is given by a label $p(v)$ for each node v. If v is reachable from s, the value $p(v)$ equals, after termination, the parent of node v in the computed shortest-paths tree. Further, a priority queue Q is used that contains $(v, d(v))$ for some nodes with tentative distance label $d(v)$ smaller than infinity.

For each node v, the value $d(v)$ is initialized to be infinity. Then, $d(s)$ is set to be 0 and s is inserted into Q. While Q is not empty, the algorithm extracts one node v with minimum distance label from Q and *relaxes* all of its outgoing edges (v, w). An edge (v, w) is relaxed as follows: If $d(w) \leq d(v) + \text{len}(v, w)$ nothing is to be done. Otherwise a tentative shortest s-w-path has been found (this path contains the node v). Hence, we set $d(w) := d(v) + \text{len}(v, w)$ and $p(w) := v$. If w is already contained in Q, its priority in the queue is updated, otherwise it is inserted into the queue. The algorithm terminates when the queue is empty. The pseudocode is listed as Algorithm 2.1.

When using a *Fibonacci-heap* as priority queue, Dijkstra's algorithm has a runtime in $O(|V| \log |V| + |E|)$. See [Jun99] and [CLRS01] for more information.

Algorithm 2.1: Dijkstra's algorithm

```
input : graph $G = (V, E, \text{len})$, source $s \in V$
uses  : priority queue $Q$
output: distance label $d(v) = \text{dist}(s, v)$ for each $v \in V$
        shortest-path tree with root $s$ given by the edges $(p(v), v)$ with $d(v) < \infty$
1  for $v \in V$ do $d(v) \leftarrow \infty$                /* Initialization Phase */
2  $d(s) \leftarrow 0$
3  $Q$.insert(s,0)
4  while not $Q$.isEmpty do                              /* Main Phase */
5      $v \leftarrow Q$.extractMin
6      for $(v, w) \in E$ do
7          if $d(v) + len(v, w) < d(w)$ then
8              $d(w) \leftarrow d(v) + len(v, w)$
9              $p(w) = v$
10             $Q$.InsertOrUpdate$(w, d(w))$
```

isEmpty	
description:	checks if S is empty
output:	true if S is empty and false otherwise

insert(u, key)	
description:	inserts (u, key) in S
precondition:	S does not contain an element (u, key') for any value key'
action:	$S := S \cup \{(u, \text{key})\}$

contains(u)	
description:	checks if S contains an element (u, key) for some value key
output:	true if there is a value key such that $(u, \text{key}) \in S$, false otherwise

extractMin	
description:	finds and outputs an element with minimum key, removes it from S
precondition:	S is not the empty set
action:	find arbitrary element (u, key) with $\text{key} = \min\{\text{key}' \mid (v, \text{key}') \in S\}$
	$S := S \setminus \{(u, \text{key})\}$
output:	u

decreaseKey(u, key)	
description:	decreases the key of element u
precondition:	S contains exactly one element (u, key') for some key' and
	$\text{key} \leq \text{key}'$ holds
action:	$S := (S \setminus \{(u, \text{key}')\}) \cup \{(u, \text{key})\}$

InsertOrUpdate(u, key)	
description:	inserts u into the queue or updates its priority
precondition:	if S contains an element (u, key') for a value key' it is $\text{key} \leq \text{key}'$
action:	if contains(u) is true then
	decreaseKey(u, key)
	otherwise
	insert(u, key)

Remove(u)	
description:	removes u from the queue
precondition:	S contains exactly one element (u, key') for some key'
action:	$S := S \setminus \{(u, \text{key}')\}$

Table 2.1: Operations of a Priority Queue.

2.3 Computational Problems and Complexity

This section shortly sketches the theory of NP-completeness and approximability. See [GJ79] for a formal and more thorough treatment of NP-completeness. The notation for optimization problems and approximation algorithms is mostly borrowed from [ACG$^+$02] which can be consulted for further information on this field.

Decision Problems and NP-completeness. In a *computational problem* we are given an *instance* and want to answer a certain *question.* In this thesis we encounter two types of computational problems: *decision problems* and *optimization problems.* In a *decision problem* the answer is either *yes* or *no.* More concrete, a decision problem $\Pi = (D_\Pi, Y_\Pi)$ is a tuple consisting of a set D_Π of *instances* together with a subset $Y_\Pi \subseteq D_\Pi$ of *yes-instances.* The elements in $D_\Pi \setminus Y_\Pi$ are called *no-instances.* We answer the decision problem on instance I by deciding whether I is a yes-instance or not.

A *polynomial-time algorithm* is an algorithm whose computation time is polynomial in the input size. We fuzzily say that the *size* of an instance I is the size of a 'reasonable encoding' of I as a string.

We call P the class of all decision problems that can be solved by a polynomial time deterministic algorithm and NP the class of all decision problems that can be solved by a polynomial-time non-deterministic algorithm. The class NP can –more intuitively– be seen as the class of all decision problems for which we can verify, in polynomial time, a potential proof of the fact that an instance I is a yes-instance.

Obviously, we have $\mathrm{P} \subseteq \mathrm{NP}$. It is still not known, but widely assumed that $\mathrm{P} \neq \mathrm{NP}$. The theory of NP-completeness gives us the power to show, for some decision problems Π, that Π lies in $\mathrm{NP} \setminus \mathrm{P}$ under the assumption $\mathrm{P} \neq \mathrm{NP}$ holds. This is done by identifying the 'most complex problems' in NP.

To compare the complexity of decision problems we use *polynomial transformations.* A polynomial transformation (or polynomial-time reduction) from a decision problem $\Pi_1 = (D_{\Pi_1}, Y_{\Pi_1})$ to a decision problem $\Pi_2 = (D_{\Pi_2}, Y_{\Pi_2})$ is a function $f : D_{\Pi_1} \to D_{\Pi_2}$ such that f can be computed by a polynomial-time deterministic algorithm and such that for all $I \in D_{\Pi_1}$ we have $I \in Y_{\Pi_1} \Leftrightarrow f(I) \in Y_{\Pi_2}$. We write $\Pi_1 \propto \Pi_2$ if there is a polynomial-time transformation from Π_1 to Π_2.

Given a polynomial-time transformation f from Π_1 to Π_2 we can solve Π_1 if we can solve Π_2: We simply decide I as we would decide $f(I)$. Hence, we consider Π_2 to be at least as difficult as Π_1 if $\Pi_1 \propto \Pi_2$. This way we only neglect polynomial time effort. Further, we consider two problems Π_1 and Π_2 to be polynomial equivalent if both, $\Pi_1 \propto \Pi_2$ and $\Pi_2 \propto \Pi_1$ hold.

We call a decision problem Π' NP-*hard*, if it is at least as difficult as all problems in NP, i.e., if, for each $\Pi \in \mathrm{NP}$, we have $\Pi \propto \Pi'$. We further call a decision problem NP-*complete* if it is NP-hard and belongs to NP. If we could solve one NP-hard problem in polynomial time we could solve all problems in NP in polynomial time. In order to show that a decision problem Π is NP-hard, we can select a known NP-hard problem Π', construct a polynomial transformation f from Π' to Π and prove that f actually is a polynomial transformation.

Note that, although explaining NP-completeness by defining decision problems on sets and polynomial reductions fuzzily by algorithms is widespread practice, we have to keep in mind that this is a simplification made for improving readability. The formally correct definitions strongly depend on the notions of *languages* and *Turing Machines.*

List of NP-hard problems. In the following, we state the NP-hard decision problems that are used for the reductions within our NP-hardness proofs. References to the corresponding NP-hardness proofs are given in [GJ79].

Problem Set Cover. Given a collection C of subsets of a finite set U and a positive integer k, is there a set cover C' of (C, U) of cardinality no more than k, i.e., a subset C' of C with $|C'| \leq k$ such that each element in U belongs to at least one member of C'?

In that context we say a set $c \in C$ covers an element u if $u \in c$. The problem remains NP-hard even if all $c \in C$ have $|c| \leq 3$. This variant of the problem is called 3-MinimumCover. For both variants we may assume that each element of U is contained in at least one set of C. The problem is solvable in polynomial time if all $c \in C$ have $|c| \leq 2$ [GJ79].

Problem Exact Cover by 3-Sets (X3C). Given a set U with $|U| = 3q$ and a collection C of 3-element subsets of U, does C contain an exact cover for U, i.e., a subcollection $C' \subseteq C$ such that each element of U occurs in exactly one member of C'?

Again we may assume that each element of U is contained in at least one set of C.

Problem VertexCover. Given an undirected graph $G = (V, E)$ and a positive integer $k \leq |V|$, is there a vertex cover of size k or less, i.e., a subset $V' \subseteq V$ of G with $|V'| \leq k$ such that for each edge $\{u, v\} \in E$ at least one of u and v belongs to V'?

Problem 3-Partition. Given a set A of $3m$ elements, a bound $B \in \mathbb{Z}_{>0}$ and a size $w_a \in \mathbb{Z}_{>0}$ for each $a \in A$ such that $B/4 < w_a < B/2$ and such that $\sum_{a \in A} w_a = mB$, can A be partitioned into m disjoint sets $A_1, A_2, \ldots, A_m$ such that, for $1 \leq i \leq m$, it is $\sum_{a \in A_i} w_a = B$?

This problem is strongly NP-hard, i.e., it remains NP-hard even if the occurring numbers are encoded unary. Note that the constraint $B/4 < w_a < B/2$ ensures that all sets A_i have cardinality 3.

Optimization Problems and Approximability. Optimization problems deal with the problem of finding a solution that minimizes or maximizes an objective function. More formally, an optimization problem is specified by the 4-tuple $\Pi = (X, S, f, \text{goal})$ where

- X is an arbitrary set of *instances*
- S is a mapping defined on X that assigns a set $S(x)$ of *feasible solutions* to each instance $x \in X$
- f is a real-valued function defined for each tuple (x, y) with $x \in X$ and $y \in S(x)$
- goal $\in \{\min, \max\}$ determines whether the problem is a minimization or maximization problem.

We call f the *objective function* of Π. A feasible solution y in $S(x)$ is an *(optimal) solution* of instance $x \in X$ if y optimizes the objective function on $S(x)$, i.e., if $f(x, y)$ is minimal or maximal (depending on goal) for $y \in S(x)$. In this case, we call $f(x, y)$ the *optimal value* of x.

We can enhance the concept of NP-hardness to optimization problems: An optimization problem Π is called NP-hard if for every decision problem $\Pi' \in$ NP, the problem Π' can be solved in polynomial time by an algorithm that uses an oracle that, for any instance of Π, returns an optimal solution along with its optimal value in constant time. As a consequence, an optimization problem is NP-hard, if an NP-hard decision problem Π' can be solved in polynomial time by such an algorithm.

Let $\Pi = (X, S, f, \text{goal})$ be an optimization problem. An *absolute approximation algorithm with maximum error k* is an algorithm that computes, for any instance $x \in X$, a solution $y \in S(x)$ such that $|f(x, y^*) - f(x, y)| \leq k$ where y^* is an optimal solution for instance x. A *constant factor approximation algorithm with approximation ratio $\alpha \geq 1$* is an algorithm that computes, for any instance $x \in X$, a feasible solution $y \in S(x)$ such that

$$\begin{cases} f(x,y)/f(y,y^*) \leq \alpha & ,\text{goal} = \min \\ f(y,y^*)/f(x,y) \leq \alpha & ,\text{goal} = \max \end{cases}$$

where y^* is an optimal solution for instance x. The class APX is the class of all optimization problems for which there exists a polynomial-time constant-factor approximation algorithm for some $\alpha \geq 1$.

2.4 Mixed-Integer Linear Programs

Mixed-Integer Linear Programs (MILPs) are mathematical models that are able to represent a wide number of combinatorial problems. Advantages of the approach are the large amount of theoretical and practical knowledge on MILPs and the availability of good black-box solvers. This section mostly uses the notation and terminology as given in [NW88].

Definitions. Solving a *mixed-integer linear programming problem* means minimizing a *linear real-valued function* of many real-valued and integer-valued variables on a domain that is given by a set of *linear constraints*. Such a problem is called *mixed* because of the simultaneous presence of integer-valued and real-valued variables. We describe a concrete instance by

$$\text{minimize } c^T x + h^T y \quad \text{such that} \quad Ax + Gy \leq b, \; x \in \mathbb{Z}^n_{\geq 0}, \; y \in \mathbb{R}^p_{\geq 0}$$

where $x = (x_1, \ldots, x_n)$ and $y = (y_1, \ldots, y_p)$ are the variables and the instance is specified by $c \in \mathbb{R}^n$, $h \in \mathbb{R}^p$, $A \in \mathbb{R}^{m \times n}$, $G \in \mathbb{R}^{m \times p}$ and $b \in \mathbb{R}^m$ for some number m. We now fix an instance $I = (c, h, A, G, b)$. The function

$$f(x, y) = c^T x + h^T y$$

is called the *objective function*. The set

$$S := \{(x, y) \mid \; Ax + Gy \leq b, \; x \in \mathbb{Z}^n_{\geq 0}, \; y \in \mathbb{R}^p_{\geq 0}\}$$

is called the *feasible region*. A tuple $(x, y) \in S$ is called a *feasible solution*. A feasible solution (x^*, y^*) is called an *optimal solution* if it minimizes the objective function on S, i.e, if

$$f(x^*, y^*) \leq f(x, y)$$

for any $(x, y) \in S$.

Expressing Problems as MILP's. Let $I = (c, h, A, G, b)$ be an instance. Each row i of $A = (a_{ij})$, $G = (g_{ij})$ and $b = (b_i)$ defines a single *constraint*:

$$\sum_{j=1}^{n} a_{ij} x_j + \sum_{j=1}^{p} g_{ij} y_j \leq b_i.$$

We alternatively describe MILP-instances by enumerating all constraints and stating the objective function. In order to represent maximization problems we can transform a linear objective function $f(x, y) = c^T x + h^T y$ to be maximized to an objective function $f'(x, y) = -c^T x - h^T y$ to be minimized. The approach is also powerful enough to express equalities. The constraint

$$\sum_{j=1}^{n} a_{ij}x_j + \sum_{j=1}^{p} g_{ij}y_j \;=\; b_i$$

can be modelled by the following two constraints

$$\begin{aligned}\sum_{j=1}^{n} a_{ij}x_j + \sum_{j=1}^{p} g_{ij}y_j \;&\leq\; b_i\\ \sum_{j=1}^{n} -a_{ij}x_j + \sum_{j=1}^{p} -g_{ij}y_j \;&\leq -b_i.\end{aligned}$$

Solvability and Restricted Problems. Let $I = (c, h, A, G, b)$ be an instance. An optimal solution of I does not necessarily exist even if the feasible region of I is not empty. We call I *unbounded* if, for any $w \in \mathbb{R}$, there is an $(x, y) \in S$ such that $c^T x + h^T y < w$. Under the assumption that all numbers occurring in I are rational and that the feasible region is not empty, one can show that I either is unbounded or has an optimal solution [NW88]. The special case $c \equiv 0$ and $A \equiv 0$ leads to the problem

$$\text{minimize } h^T y \;\text{ such that }\; Gy \leq b,\; y \in \mathbb{R}^p_{\geq 0}$$

and is called *linear program (LP)*. This can be solved in polynomial time, for example by the *ellipsoid algorithm*. An experimentally fast algorithm for solving linear programs is the *simplex algorithm*. However, this algorithm has exponential worst-case runtime. The special case $h \equiv 0$ and $G \equiv 0$ leads to the problem

$$\text{minimize } c^T x \;\text{ such that }\; Ax \leq b,\; x \in \mathbb{Z}^n_{\geq 0}$$

and is called *integer linear program (ILP)*. This problem is NP-hard already for very special cases. See [NW88] for an overview on solution approaches for MILP's, ILP's and LP's.

Chapter 3

Preprocessing Speedup-Techniques is Hard

During the last years, preprocessing-based techniques have been developed to compute shortest paths between two given points in road networks. These *speedup-techniques* make the computation a matter of microseconds even on huge networks, as opposed to seconds required by Dijkstra's algorithm. While there is a vast amount of experimental work in the field, there is still large demand for theoretical foundations. The preprocessing phases of most speedup-techniques leave open some degree of freedom which, in practice, is filled in a heuristical fashion. Thus, for a given speedup-technique, the problem arises of how to fill the according degree of freedom optimally. Until now, the complexity status of these problems has been unknown. In this chapter, we answer this question by showing NP-hardness for the recent techniques.

3.1 Motivation

Computing shortest paths in graphs is used in many real-world applications like route-planning in road networks or for finding good connections in railway timetable information systems. In general, Dijkstra's algorithm computes a shortest path between a given source and a given target. Unfortunately, the algorithm is slow on huge datasets. Therefore, it cannot be directly used for applications like car navigation systems or online route-planners that require an instant answer to a source-target query.

Often, this problem is coped with by dividing the computation of the shortest paths into two stages. In the *offline stage*, some data is precomputed that is used in the *online stage* to answer a source-target query heuristically faster than Dijkstra's algorithm. Such an approach is called a *speedup-technique* (for Dijkstra's algorithm). During the last years, speedup-techniques have been developed for road networks (see [DSSW09] for an overview), that make the shortest path computation a matter of microseconds even on huge road networks consisting of millions of nodes and edges.

Usually, the offline stage leaves open some degree of freedom, such as the choice of how to partition a graph or of how to order a set of nodes. The decision taken to fill the respective degree of freedom has direct impact on the search space of the online stage and therefore on the runtime of a query. Currently, these decisions are made in a purely heuristical fashion. A common trade-off is between preprocessing time/space and query time/search space. Practitioners in the field usually compare their results by absolute query times, size of the search space, size of the preprocessed data and by the time needed for the preprocessing phase. In this chapter we show the NP-hardness of performing the offline stage such that the average search space of the query is optimal. For each technique we demand that the size of the preprocessed data is bounded by a given parameter. In practice, the basic technique can be enriched by various heuristic improvements. We do not consider such improvements and stick to the basic core of each technique. However, some interesting details are treated separately. This implies that, for the sake of simplicity, some techniques are slightly altered. The techniques considered are

- ALT [GH05, GW05]
- Arc-Flags [Lau04, KMS05, MSS$^+$05, MSS$^+$06, Sch06a, HKMS06, HKMS09]
- SHARC [BD08, BD09, Del09, BDGW10]
- Multilevel Overlay Graph [SWZ02, Sch05, HSW06, SS07, HSW08, Hol08]
- Contraction Hierarchies [GSSD08].

We left out

- Geometric Containers [SWW99, SWW00, WW03, WWZ05, Wil05]
- Highway Hierarchies [SS05, SS06a]
- Reach-Based Pruning [Gut04, GKW06, GKW07, GKW09]

as their offline stages merely contain tuning parameters but no real degree of freedom. However, two interesting side-aspects of Reach-Based Pruning are included. We further did not work on

- Transit Node Routing [SS06b, Mül06, BFM$^+$07, DHM$^+$09, BFM09]

as this is a framework for which also parts of the query algorithm need to be specified.

An Additional Question. We further raise the question of how good a given speedup-technique actually can get. We give a lower bound on any guarantee for the average CONTRACTION HIERARCHIES-search space size on sparse graphs.

Related Work. There is a huge amount of work on speedup-techniques. An overview of experimental work can be found in [WW07b, DSSW09]. There is large demand for a theoretical foundation for the field and there is only little theoretical work: Part of this chapter has previously been published in [BCK$^+$10a, BCK$^+$10b]. Based on these papers, the student's thesis [Fuc10] considers the preprocessing phase of the ALT-algorithm. A worst-case model for the search space is proposed, ILP-based approaches and a connection to the maximum coverage problem are given. The student's thesis [Col09] considers the preprocessing phase of Contraction Hierarchies. The NP-hardness of the preprocessing phase is proven and a lower bound for the search space size is given. The contents of [Col09] mostly are covered by [BCK$^+$10a, BCK$^+$10b] and this thesis.

In [BDDW09, BDD$^+$10] results are given for the SHORTCUT PROBLEM, a problem related to the technique of inserting *shortcuts* to the underlying graph. Chapter 4 covers these results. Recently, a graph generator for road networks was given in [AFGW10]. There, graphs evolving from this generator are shown to exhibit a property called *low highway dimension*. For graphs with this property, a special preprocessing technique is proposed and runtime guarantees for Reach-Based Pruning, Contraction Hierarchies, Transit Node Routing and SHARC using that preprocessing technique are given.

3.2 Problem Statement

There are two types of speedup-techniques. *Unidirectional* techniques base on Dijkstra's algorithm and find the target by conducting *one* search starting at the source. *Bidirectional* approaches rely on *Bidirectional Search* and simultaneously perform a search starting at the source *and* a backward search starting at the target. In the following we describe both types followed by a generic description of the problem considered in this chapter. The concrete problem is stated separately for each technique in the according section. Afterwards we report on some small but important technical details.

Setting. Let $G = (V, E, len)$ denote a directed, weighted graph with positive length function $len : E \to \mathbb{R}_{>0}$. In this chapter, we consider techniques that answer *s-t-queries* in G, i.e., they compute the distance from *source s* to *target t* for arbitrary nodes s and t in V.

We have online route-planning software in mind that has to answer a large number of s-t-queries. Hence, we allow an *offline phase* in which the graph is already given, but source and target are unknown. The output of the offline phase is additional data that can be accessed during the *online phase*. In this phase source s and target t are known and $\text{dist}(s,t)$ is computed with the help of the additional data. Obviously, one could gain optimal query time by simply precomputing all distances in the graph. However, this is often impractical for real-world input because of the large size of the underlying networks. Each considered technique is based on Dijkstra's algorithm and could be extended such that also a shortest s-t-path is returned. See [DSSW09] for a description of such enhancements.

Unidirectional Speedup-Techniques. We study two unidirectional techniques: ALT and Arc-Flags. For both approaches, the online phase (i.e., the query) basically works like Dijkstra's algorithm and both are *label-setting*: Each node v is inserted into the queue only once. After extracting v from the queue, the distance label $d(v)$ is correct and v is not inserted into Q again. We call a node v *visited* if at least one incoming edge of v has been relaxed, i.e., if $d(v) < \infty$. We call a visited node *settled*, if it has been extracted from Q. As we focus on s-t-queries, we can stop after the target t has been extracted from the queue. The *search-space* of an s-t-query is the set of nodes settled up to the point when t gets settled, including t.

However, the main speedup compared with Dijkstra's algorithm must be attributed to additional improvements: In order to direct the search in the direction of the target, ALT applies a different priority when extracting nodes from Q. Arc-Flags prune the search space by not relaxing edges that, in the preprocessing phase, have been recognized as useless for the given query.

Bidirectional Search. This approach simultaneously starts a Dijkstra's search rooted at s on G (the *forward search*) and one rooted at t on the reverse graph $\overleftarrow{G}$ (the *backward search*). Whenever a node has been settled it has to be decided if the algorithm should change to the opposite search. A simple approach is to swap the direction every time a node is settled. The *distance balanced* bidirectional search changes to the other direction if, and only if, the minimal distance label of nodes in the queue is greater than the minimal distance label of nodes in the contrary queue. The algorithm can stop when one node is settled in both directions. Finally, $\text{dist}(s,t) = \min\{\text{dist}(s,v) + \text{dist}(v,t)\}$ over all nodes v, that get visited from both directions. Note that the first node v settled in both directions is not necessarily part of the shortest s-t-path. This can be seen from the following simple example: $G = (\{s,t,v\}, \{\{s,t\},\{s,v\},\{t,v\}\})$ with $\text{len}(\{s,t\}) = 10$ and $\text{len}(\{s,v\}) = \text{len}(\{t,v\}) = 6$.

The worst-case behavior of this approach is not better than Dijkstra's algorithm.

Algorithm 3.1: Query of a bidirectional speedup-technique

```
input  : graph $G = (V, E, \mathrm{len})$, source $s \in V$, target $t \in V$,
         additional preprocessed data $\alpha$
output: $\mathrm{dist}(s, t)$
1 $d^+ \leftarrow$ unidirectionalPrunedSearch($G, s, \alpha$)          /* forward search */
2 $d^- \leftarrow$ unidirectionalPrunedSearch($\overleftarrow{G}, t, \alpha$)          /* backward search */
3 output $\min\{d^+(v) + d^-(v) \mid v \in V\}$
  /* search space is the union of the search spaces of forward and
     backward search, count nodes contained in both searches twice    */
```

However, on *usual* inputs, bidirectional search reduces the search space by a factor of 2 compared with Dijkstra's algorithm.

Bidirectional Speedup-Techniques. We study the following bidirectional techniques: Reach-Based Pruning, Multilevel-Overlay Graph and Contraction Hierarchies. The core of the online phase of these approaches is an adapted bidirectional search. The forward and the backward search are changed such that some edges or nodes are not considered. Thus, part of the graph is pruned from the search space. The main speedup results from such prunings and it is reasonable to relax the stopping criterion for our theoretical model. Consequently, we later see that forward and backward search are completely independent from each other. Hence, the strategy of how to swap between the searches does not matter anymore and we can execute both searches sequently. We apply this query scheme for each bidirectional technique. Pseudocode for the main routine is given as Algorithm 3.1, the individual pruned unidirectional searches are stated for each technique separately in the respective sections.

Similar to the unidirectional case, a node v is called *visited* in one direction as soon as an incoming edge of v is relaxed in the according search and *settled* in this direction when it has been extracted from the queue of this direction. For bidirectional speedup-techniques, the *search-space* of an s-t-query is the union of the search-spaces of forward and backward search of this query. We consider the search space to be a multiset, i.e., when computing the size of the search space we count nodes that get settled in both directions twice.

Problem Statement. The offline phase of each technique has a particular degree of freedom such as the choice of how to order a set of nodes or the choice of how to partition the underlying graph. This affects the output of the offline phase and hence the additional data α used in the online phase.

For a given technique, we write $V_\alpha(s, t, G)$ for the search space of an s-t-query in graph G, when having the additional data α as input. We omit G and write $V_\alpha(s, t)$ in case the choice of the underlying graph is clear. Our aim is to fill the according degree of freedom, such that the average search space of a query becomes minimal. We choose source and target uniformly at random and hence want to compute the additional data α such that $\sum_{s,t\in V} |V_\alpha(s, t)|$ is minimized.

When working with bidirectional techniques, we use an equivalent formulation of the seach space size: We denote by $V^+_\alpha(z)$ and $V^-_\alpha(z)$ the search space of forward and backward search starting at z, respectively. As both searches are completely independent from each other, we can make the following transformation:

$$\sum_{s,t\in V} |V_\alpha(s,t)| = \sum_{s,t\in V} |V^+_\alpha(s)| + |V^-_\alpha(t)| = |V| \sum_{z\in V} \left(|V^+_\alpha(z)| + |V^-_\alpha(z)|\right) \tag{3.1}$$

The choices undertaken in the preprocessing phase also affect the size of the preprocessed data. For each technique, we demand that the size of the preprocessed data is bounded by a given parameter.

Breaking Ties in a Priority Queue. There are many possibilities to break ties when extracting nodes from the queue. We follow an idea of Goldberg and Harrelson [GH05] that is helpful for theoretical considerations: Throughout this work, we additionally identify each node uniquely with an integer between 1 and $|V|$. Among all nodes with minimal priority in the queue, the smallest integer gets extracted first.

Shortcuts. Given is a graph $G = (V, E, \text{len})$. A shortcut is an additional edge (u, v) that is inserted into G such that $\text{len}(u, v)$ equals the distance from s to v. See Chapter 4 for more results on shortcuts.

Definition (Shortcut Assignment). Let $G = (V, E, \text{len})$ be a graph. A *shortcut assignment* for G is a set $E' \subseteq (V \times V) \setminus E$ such that, for any (u, v) in E', it is $\text{dist}(u, v) < \infty$. The notation $G[E']$ abbreviates the graph G with the shortcut assignment E' added, i.e., the graph $(V, E \cup E', \text{len}')$ where $len' : E \cup E' \to \mathbb{R}_{>0}$ equals $dist(u, v)$ if $(u, v) \in E'$ and equals $len(u, v)$ otherwise.

Some speedup-techniques use shortcuts to reduce the search space size, a detailed description is given separately for each technique.

Requirements on the Input Graph. Consider the following situation: We are given an edge (u, v) in a graph G such that the shortest u-v-path does not contain (u, v). After inserting the shortcut (u, v), the graph $G[\{(u, v)\}]$ is a multi-graph. Hence, for simplicity we demand that $\text{len}(u, v) = \text{dist}(u, v)$ for each $(u, v) \in E$. This can easily be assured by deleting all edges with $\text{len}(u, v) > \text{dist}(u, v)$ in a preprocessing step.

A Repeating Pattern. We often face the following technical task: We consider Dijkstra's algorithm and are given a set $T \subseteq V$ such that $T = \{t \in V \mid \text{dist}(s, t) = c\}$ for a number c. Our aim is to compute the sum $\sum_{t \in T} |V(s, t)|$ of the search-space sizes of all s-t-queries with $t \in T$.

For each node $t \in T$, we have to settle all nodes in $\{v \in V | \, \text{dist}(s, v) < \text{dist}(s, t)\}$ before we can settle t. We remember that, when deciding which node to settle next, ties are broken according to some predefined order on the vertices. Hence, the part of nodes in T on the overall search-space size is $1 + 2 + \ldots + |T| = |T|(|T| + 1)/2$. Summarizing, we can decompose

$$\sum_{t \in T} |V(s, t)| = |T| \cdot \left| \left\{ v \in V \mid \text{dist}(s, v) < \text{dist}(s, t) \right\} \right| + |T|(|T| + 1)/2 \; . \tag{3.2}$$

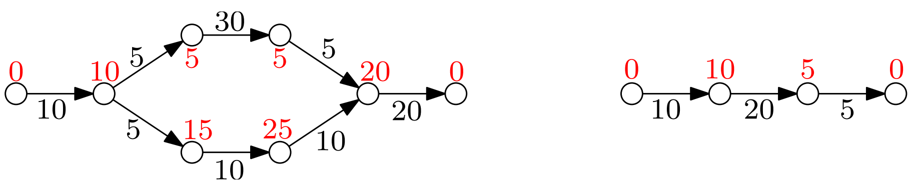


Figure 3.1: Reach values of a sample graph and a sample path. Edge labels represent edge lengths, node labels represent reach values.

3.3 Reach-Based Pruning

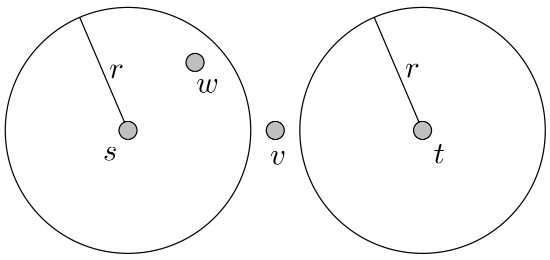
Reach is a centrality measure introduced by Gutman [Gut04]. We use the version of Goldberg et al. [GKW06, GKW07, GKW09]. In this section, let $\mathrm{SP}(G)$ denote the set of all shortest paths in G. The reach $\mathcal{R}_P(v_i)$ of a node v_i with respect to a path $P = (v_1, \dots, v_k)$ with $v_i \in P$ is

$$\mathcal{R}_P(v_i) := \min\{\mathrm{len}(v_1, \dots, v_i), \mathrm{len}(v_i, \dots, v_k)\}.$$

The reach $\mathcal{R}(v)$ of a node with respect to a graph G is

$$\mathcal{R}(v) := \max_{\{P \in \mathrm{SP}(G) \mid \ v \in P\}} \mathcal{R}_P(v).$$

See Figure 3.1 for an example. For ease of notation, we consider a single vertex to be a path of length 0. The reach of a node is high, if it lies in the middle of a long shortest path.



Sometimes, nodes of low reach can be pruned from a bidirectional search. Consider the situation to the left. The nodes v and w both have reach $\mathcal{R}(v) = \mathcal{R}(w) = r$. As $\mathrm{dist}(s, v) > \mathcal{R}(v)$ and $\mathrm{dist}(v, t) > \mathcal{R}(v)$ we do not have to consider v in any direction: If there would be a shortest s-t-path containing v, this path would effect in $\mathcal{R}(v) \geq \min\{\mathrm{dist}(s, v), \mathrm{dist}(v, t)\}$ which is a contradiction.

We further do not have to consider w in the search starting at t if we assure that w is included in the search starting at s: We will see that we stay correct if we do not consider a node w in the search with root $x \in \{s, t\}$, if $w's$ distance from x is larger than $\mathcal{R}(w)$.

Query. There exist different variants of how to use reach values for pruning the search space of a bidirectional search, all of them sharing the same main idea. The variant described here is called the *self-bounding query.* In practice the approach is mixed with other ingredients like ALT, contraction and the computation of upper bounds for reach values which we do not consider here.

The query is a standard query of a bidirectional speedup-technique (Algorithm 3.1). The according undirectional pruned search is Dijkstra's algorithm with the difference, that we only insert or change nodes w in the queue, for which $\mathcal{R}(w) \geq d(w)$. The pseudocode is given as Algorithm 3.2. Whenever we give pseudocode for a speedup-technique, differences with Dijkstra's algorithm are marked grey.

Correctness. Given is an s-t-query. If $\mathrm{dist}(s, t) = \infty$ the correctness follows directly. Now let $P = (v_1, \dots, v_k)$ be a shortest s-t-path. To show the correctness of the approach,

Algorithm 3.2: Unidirectional Pruned Search of Reach-Based Pruning Query

```
input  : graph $G = (V, E, \text{len})$, node $x \in V$, reach values $\mathcal{R}()$
output: distance label $d()$
for $v \in V$ do $d(v) \leftarrow \infty$                    /* Initialization Phase */
$d(x) \leftarrow 0$ ; $Q$.insert(x,0)
while not $Q$.isEmpty do                                  /* Main Phase */
    $v \leftarrow Q$.extractMin       /* $v$ is now contained in the search space */
    for $(v, w) \in E$ do
        if $d(v) + len(v, w) < d(w)$ then
            $d(w) \leftarrow d(v) + len(v, w)$
            if $\mathcal{R}(w) \geq d(w)$ then $Q$.InsertOrUpdate($w, d(w)$)
```

we assure that there is a node w in G such that $d^+(w) + d^-(w) = \text{dist}(s,t)$. As then $d^+(w) < \infty$ and $d^-(w) < \infty$ we know that v is visited from both directions. For each node v in P it is $\mathcal{R}(v) \geq \mathcal{R}_P(v)$. Hence, we can split P 'in the middle' into $P_1 = (v_1, \ldots, v_l)$ and $P_2 = (v_{l+1}, \ldots, v_k)$ such that for each node $v \in P_1$ it is $\mathcal{R}(v) \geq \text{dist}(s,v)$ and for each node $v \in P_2$ it is $\mathcal{R}(v) \geq \text{dist}(v,t)$. Accordingly, all nodes in P_1 get settled in the forward search and all nodes in P_2 get settled in the backward search. Consequently, v_{l+1} gets visited by the forward search which suffices to show the correctness.

3.3.1 Search-Space Minimal Reach.

In case shortest paths are not unique the technique still computes correct distances even if only considering one shortest path for each source-target pair. This can be seen in the above proof of correctness. When regarding less shortest paths, the resulting reach values may decrease. This can lead to pruning more nodes and hence the search space may shrink. The Problem MinReach is that of choosing these shortest paths such that the resulting average search-space becomes minimal. More formally, we choose a set $\mathcal{P}$ of shortest paths and compute $\mathcal{R}(v)$ by $\max_{\{P \in \mathcal{P} \mid \, v \in P\}} \mathcal{R}_P(v)$. We denote by $V_{\mathcal{P}}(s,t,G)$ the search space of the s-t-reach-query using these reach-values as input for the search.

Problem MinReach. Given a graph $G = (V, E, \text{len})$, choose $\mathcal{P} \subseteq \text{SP}(G)$ such that $\mathcal{P}$ contains one shortest s-t path for each pair of nodes $s, t \in V$ with $\text{dist}(s,t) < \infty$ and such that $\sum_{s,t \in V} V_{\mathcal{P}}(s,t,G)$ is minimal.

Theorem 1. Problem MinReach is NP-hard (even for directed acyclic graphs).

Proof. We make a reduction from Exact Cover by 3-Sets (X3C, page 11). Given an X3C-instance (U, C) with $|U| = 3q$ we may assume that each element in U is contained in at least one set in C. We construct a MinReach-instance $G = (V, E, \text{len})$ as follows, see Figure 3.2 for a visualization: $V = \{a\} \cup C \cup U$ where a is an additional vertex. There is an edge (a, c) for each $c \in C$. There is an edge $(c, u) \in C \times U$ if, and only if $u \in c$. All edge lengths are 1. The construction is polynomial.

It is $\mathcal{R}(u) = 0$ for $u \in \{a\} \cup U$ as these nodes are contained in paths only as start- or end-nodes. Hence, these nodes only get settled as start nodes from the forward search or as target nodes from the backward search. Given the set $\mathcal{P}$, we denote the search space starting at node z by $V^+_{\mathcal{P}}(z)$ and $V^-_{\mathcal{P}}(z)$ for forward and backward search on G,

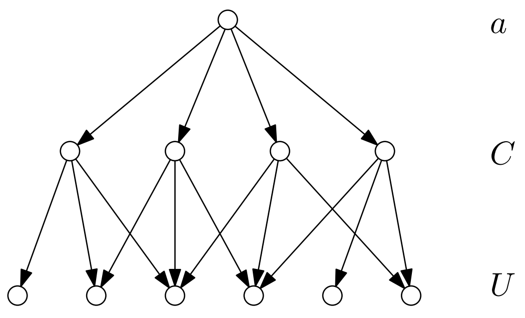


Figure 3.2: Graph G constructed from the X3C-instance $\{1,2,3\},\{2,3,4\},\{3,4,6\},\{4,5,6\}$.

respectively. It is

$$\begin{aligned}\sum_{s\in U\cup C}|V^+(s)| &= |U\cup C|\\ \sum_{t\in\{a\}\cup C}|V^-(t)| &= |C|+1\end{aligned}$$

as the corresponding unidirectional searches only settle their start node. We use Equation 3.1 and decompose

$$\sum_{s,t\in V}|V_\mathcal{P}(s,t)| = |V|\Big(|V^+(a)| + \underbrace{\sum_{s\in U\cup C}|V^+(s)|}_{=|U\cup C|} + \underbrace{\sum_{t\in\{a\}\cup C}|V^-(t)|}_{=|C|+1} + \sum_{t\in U}|V^-(t)|\Big)$$

Claim. There is a set $\mathcal{P}$ such that

$$\sum_{s,t\in V}V_\mathcal{P}(s,t) \le |V|\Big((1+q)+|U\cup C|+|C|+1+2|U|\Big) \tag{3.3}$$

if and only if there is an exact cover for (U,C).

'*If*'. When computing the reach-values of nodes in C we only have to consider paths that start with a and end in U because paths consisting of only one edge do not contribute to reach-values greater than 0. Let $C'\subseteq C$ be an exact cover of (U,C). Further let $C'(u)$ denote the $c'\in C'$ with $u\in c'$. We set $\mathcal{P}=\{(a,C'(u),u)|\ u\in U\}$. Then, for each $c\in C$ we have $\mathcal{R}(c)=1$ if there is a path (a,c,u) in $\mathcal{P}$ and $\mathcal{R}(c)=0$ otherwise. Hence, $|V^+(a)|=1+q$ and $|V^-(u)|=2$ for each $u\in U$. This yields the claimed bound.

'*Only if*'. Let $\mathcal{P}\subseteq \mathrm{SP}(G)$ be such that Equation 3.3 holds. We show that $C'=\{c\in C|\ (a,c,u)\in\mathcal{P}\}$ is an exact cover of (U,C). As $\mathcal{P}$ has to include one shortest a-u path for each $u\in U$ we know that C' covers U. With the above decomposition of the search-space we know that $|V^+(a)|+\sum_{t\in U}|V^-(t)|\le 1+q+2|U|$. It is

$$V^+(a)=\{a\}\cup\{c\in C|\ \mathcal{R}(c)\ge 1\}=\{a\}\cup C'$$

and, for $u\in U$,

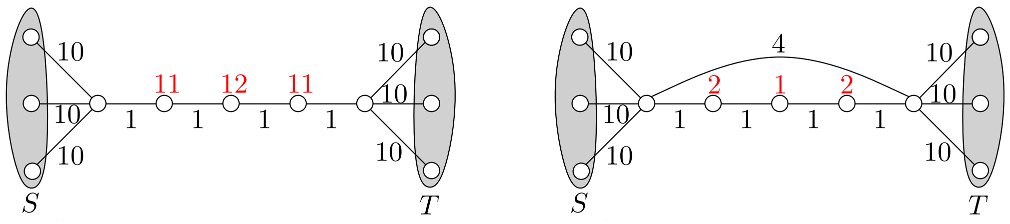
$$\begin{aligned}V^-(u) &= \{u\}\cup\{c\in C|\ \mathcal{R}(c)\ge 1, u\in c\}\\ &= \{u\}\cup\{c\in C|\ (a,c,u)\in\mathcal{P}\}\ge 2.\end{aligned}$$

Hence $|C'|\le q$ which implies the claim.

Summary. We have shown that the decision variant of problem MinReach that asks if it is possible to reach a certain search-space size is NP-hard. Hence, also the optimization variant, i.e., problem MinReach is NP-hard. □

3.3.2 External Shortcuts for Reach-Based Pruning.

Given is a graph $G = (V, E, \mathrm{len})$. A shortcut is an additional edge (u, v) that is inserted into G such that $\mathrm{len}(u, v)$ equals the distance from u to v (see Section 3.2, page 19 for a proper definition). The notation $G[E']$ abbreviates the graph G with the set E' of shortcuts added. Shortcuts can be used to reduce reach values and hence to diminish the search space. Consider the left-hand graph in the following example, node labels give the reach values, edge labels give the edge weights.



When performing an s-t-query with $s \in S$ and $t \in T$ we have to settle the three nodes in the middle. We compute reach values as for problem MinReach: For each s-t-pair we consider only one shortest s-t-path. The graph to the right contains one additional shortcut. Hence, reach values of the center nodes can be chosen smaller and the center nodes are not contained in such an s-t-query.

Given an input graph G and a parameter k, we consider the following scenario: The preprocessing phase is split into two stages. In Stage 1, we are allowed to insert a set $\mathcal{S}$ of k shortcuts into G. This yields the graph $G[\mathcal{S}]$. After Stage 1, the remaining part of the preprocessing phase is determined. Stage 2 is problem MinReach on $G[\mathcal{S}]$, i.e., we choose a subset of the shortest paths in the graph such that the resulting reach values minimize the average search space of the online phase. We consider two different variants of Stage 2. Accordingly we assume that we have two different means to solve problem MinReach:

- An oracle opt, that returns an optimal solution $\mathrm{opt}(G[\mathcal{S}]) \subseteq \mathrm{SP}(G[\mathcal{S}])$ of MinReach-instance $G[\mathcal{S}]$ (Variant 1).
- A heuristic algorithm apx that returns an arbitrary feasible solution $\mathrm{apx}(G[\mathcal{S}]) \subseteq \mathrm{SP}(G[\mathcal{S}])$ that, for each connected s-t-pair, contains one *edge-minimal* shortest s-t-path (Variant 2).

We show for both variants, that it is NP-hard to choose $\mathcal{S}$ such that the objective function of MinReach-instance $G[\mathcal{S}]$ is minimized.

Problem ExtShortcutsReach. Given a graph $G = (V, E, \mathrm{len})$ and a positive integer k, find a shortcut assignment $\mathcal{S}$ of cardinality k, such that

- $\sum_{s,t \in V} V_{\mathrm{opt}(\mathcal{S})}(s, t) := \sum_{s,t \in V} V_{\mathrm{opt}(G[\mathcal{S}])}(s, t, G[\mathcal{S}])$ (Variant 1)
- $\sum_{s,t \in V} V_{\mathrm{apx}(\mathcal{S})}(s, t) := \sum_{s,t \in V} V_{\mathrm{apx}(G[\mathcal{S}])}(s, t, G[\mathcal{S}])$ (Variant 2)

is minimal

Theorem 2. Both variants of problem ExtShortcutsReach are NP-hard (even for directed acyclic graphs).

Proof. We make a reduction from Exact Cover by 3-Sets (X3C, page 11). Given an X3C-instance (U, C) with $|U| = 3q$ we may assume that each element in U is contained in at least one set in C. We construct an instance $(G = (V, E, \mathrm{len}), k = q)$ of ExtShortcutsReach

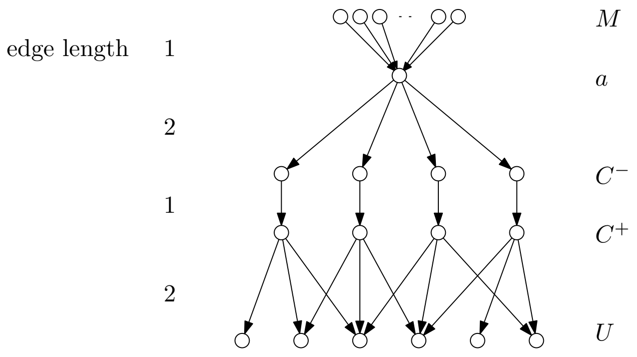


Figure 3.3: Graph G constructed from the X3C-instance $\{1,2,3\},\{2,3,4\},\{3,4,6\},\{4,5,6\}$.

as follows, see Figure 3.3 for a visualization: The set V consists of two nodes c^- and c^+ for each $c \in C$, one node u for each $u \in U$, one additional node a and an additional set M with $|M|$ to be specified later. We denote by C^- the set $\{c^- \mid c \in C)$ and by C^+ the set $\{c^+ \mid c \in C)$. There is an edge (c^+, u) with length 2 if $u \in c$. Further, there are edges (a, c^-) with length 2 and (c^-, c^+) with length 1 for each $c \in C$. Moverover, there is an edge (m, a) with length 1 for each $m \in M$. We set $k = q$. The transformation is polynomial as $|M|$ will be polynomial in the input size.
We abbreviate the search spaces of the unidirectional pruned searches in the augmented graph as follows:

$$\begin{aligned} V^-_{\mathrm{opt}(\mathcal{S})}(z) &:= V^-_{\mathrm{opt}(G[\mathcal{S}])}(z, G[\mathcal{S}]) & \qquad V^-_{\mathrm{apx}(\mathcal{S})}(z) &:= V^-_{\mathrm{apx}(G[\mathcal{S}])}(z, G[\mathcal{S}]) \\ V^+_{\mathrm{opt}(\mathcal{S})}(z) &:= V^+_{\mathrm{opt}(G[\mathcal{S}])}(z, G[\mathcal{S}]) & \qquad V^+_{\mathrm{apx}(\mathcal{S})}(z) &:= V^+_{\mathrm{apx}(G[\mathcal{S}])}(z, G[\mathcal{S}]). \end{aligned}$$

We leave out the subscript apx/opt if the choice of the variant does not matter. We use Equation 3.1:

$$\sum_{s,t \in V} V_{\mathcal{S}}(s,t) = \underbrace{|V| \sum_{z \in V \setminus M} \left(V^-_{\mathcal{S}}(z) + V^+_{\mathcal{S}}(z)\right)}_{=:\alpha} + |V| \sum_{z \in M} (\underbrace{V^-_{\mathcal{S}}(z)}_{=|1|} + V^+_{\mathcal{S}}(z)).$$

It is $\mathcal{R}(m) = 0$ for $m \in M$ as these nodes are at most start-nodes of shortest paths in the graph. Hence searches starting in $V \setminus M$ do not settle nodes in M and we have $\alpha \leq 2|V \setminus M|^2$. We call a shortcut assignment $\mathcal{S}$ *set covering* if $\mathcal{S}$ contains, for each $u \in U$, a shortcut (a, c^+) such that $u \in c$. If $\mathcal{S}$ is set-covering it is $\mathcal{S} \subseteq \{a\} \times C^+$ as $\mathcal{S}$ contains only q shortcuts and these have to cover the $3q$ nodes in U.

The proof outlines as follows: Obviously, a set covering shortcut assignment of (G, q) induces a set cover of (U, C) and vice versa. We give an upper bound for the search space size of a set covering shortcut assignment. Then, we give a lower bound for the search space size of a shortcut assignment that 'cannot be transformed to a set cover of (U, C)'. Finally, we show that, for $|M|$ big enough, both classes can be separated by a bound on the search space size and establish the desired relationship between solutions of (C, U) and (G, q).

Claim. Let $\mathcal{S}$ be set-covering. Then $\sum_{m \in M} V^+_{\mathrm{apx}(\mathcal{S})}(m) \leq 2|M|$.

Let m be in M. We proof the claim by showing that $reach(x) < \mathrm{dist}(m, x)$ for $x \in C^+ \cup C^- \cup U$. Hence $V^+(m) \subseteq \{m, a\}$ and it immediately follows the claim. It is $\mathcal{R}(u) = 0$ for $u \in U$ as these nodes are at most end-nodes of shortest paths. With $\mathrm{dist}(m, u) = 6$ we have $V^+_{\mathrm{apx}(\mathcal{S})}(m) \cap U = \emptyset$. Further, for $c^+ \in C^+$ we have $\mathcal{R}(c^+) \leq 2$ and $\mathrm{dist}(m, c^+) = 4$. Hence $V^+_{\mathrm{apx}(\mathcal{S})}(m) \cap C^+ = \emptyset$. Remember that $\mathrm{apx}(\mathcal{S})$ always chooses edge-minimal shortest paths. An edge-minimal shortest path P starting in M and ending in U contains a shortcut in $\mathcal{S}$ as $\mathcal{S}$ is set-covering. Hence, the shortest paths from M to U chosen by $\mathrm{apx}(\mathcal{S})$ do not contain any node in C^-. Accordingly, $\mathcal{R}(c^-) \leq 2 < \mathrm{dist}(m, c^-)$ for c^- in C^-. Hence $V^+_{\mathrm{apx}(\mathcal{S})}(m) \subseteq \{m, a\}$.

Corollary. This implies that $\sum_{m \in M} V^+_{\mathrm{opt}(\mathcal{S})}(m) \leq 2|M|$ if $\mathcal{S}$ is set-covering.

Claim. Assume $|M| > k$. Let $\mathcal{S}$ and $u^* \in U$ be such that $(a, u^*) \notin \mathcal{S}$ and such that for all $c \in C$ with $u^* \in c$, we have $(a, c^+) \notin \mathcal{S}$. Then $\sum_{m \in M} V^+_{\mathrm{opt}(\mathcal{S})}(m) \geq 3|M|$.

We show that, for each $m \in M$, it is $V^+_{\mathrm{opt}(\mathcal{S})}(m) \supseteq \{m, a, c^-_*\}$ for an $c^-_* \in C^-$ which proofs the claim. As $|M| > k$ we have at least one node $m' \in M$ such that $(m', v) \notin \mathcal{S}$ for all $v \in V$. Hence, (m', a, c^-) is the only shortest path for $c^- \in C$ and $\mathcal{R}(a) \geq 1$. As $\mathrm{dist}(m, a) = 1$, the node a gets settled from each $m \in M$. Further, a shortest m'-u^*-path in $G[\mathcal{S}]$ starts with (m', a, c^-_*) for a $c^-_* \in C^-$. Hence, $\mathcal{R}(c^-_*) \geq 3$ and c^-_* gets settled from all $m \in M$. Summarizing, for each $m \in M$, we have that $V^+_{\mathrm{opt}(\mathcal{S})}(m) \supseteq \{m, a, c^-_*\}$ which shows the claim.

Corollary. With the requirements of the last claim follows that $\sum_{m \in M} V^+_{\mathrm{apx}(\mathcal{S})}(m) \geq 3|M|$.

Claim. We specify $|M| = \max\{k, 2|V \setminus M|^2\} + 1$. Then, there is a shortcut assignment $\mathcal{S}$ such that

$$\sum_{s,t \in V} V_{\mathcal{S}}(s, t) \leq |V| \Big(2|V \setminus M|^2 + |M| + 2|M| \Big)$$

if and only if there is an exact cover for (U, C).

Let C' be an exact cover of (U, C). Then $\{(a, c^+)|\ c \in C'\}$ is set-covering and the bound on the search-space holds with the above claims. On the other hand let $\mathcal{S}^*$ be such that

$$\sum_{s,t \in V} V_{\mathcal{S}^*}(s, t) \leq |V| (2 \cdot |V \setminus M|^2 + |M| + 2|M|) \ .$$

With the last claim we know that for each $u \in U$ there must be either a shortcut $(a, u) \in \mathcal{S}^*$ or a shortcut (a, c^+) with $u \in c$ as otherwise

$$\sum_{s,t \in V} V_{\mathcal{S}}(s, t) \geq |V| \Big(|M| + 3|M| \Big)$$

with contradiction to the assumption We gain a shortcut assignment $\mathcal{S}'$ out of $\mathcal{S}^*$ by copying all shortcuts of the form (a, c^+) in $\mathcal{S}^*$ and taking, for each shortcut of the form $(a, u) \in \mathcal{S}^*$, one arbitrary shortcut (a, c^+) with $u \in c$. The set $\{c|\ (a, c^+) \in \mathcal{S}'\}$ is a cover of (U, C) of size q.

Summary. We have shown that the decision variant of problem ExtShortcutsReach that asks if it is possible to reach a certain search-space size is NP-hard. Hence, also the optimization variant, i.e., problem ExtShortcutsReach is NP-hard. □

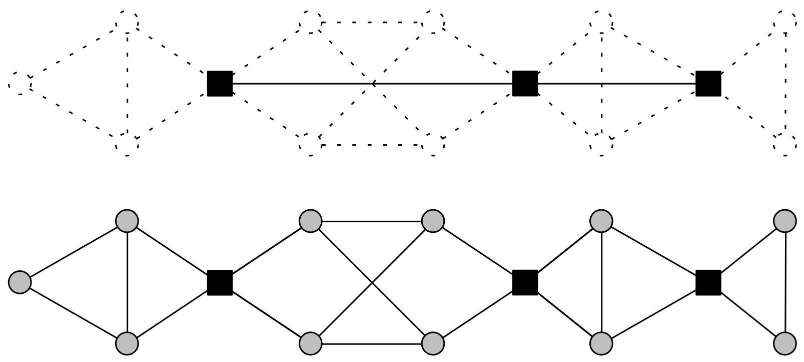


Figure 3.4: Visualization of a 2-level multilevel overlay graph. Vertices in V_1 are drawn quadratic. The graph G_0 is given at the bottom, the graph G_1 at the top.

3.4 Multilevel Overlay Graph

This bidirectional, hierarchical approach has a long history [SWZ02, Sch05, HSW06, SS07, HSW08, Hol08] and sometimes is also called Highway Node Routing. Given is the input graph $G = (V, E, \text{len})$ and a *number of levels* $L + 1$. The degree of freedom is to choose a sequence $V := V_0 \supseteq V_1 \supseteq \ldots \supseteq V_L$ of subsets of V. The preprocessing phase then computes a sequence $(G_0, G_1, \ldots, G_L)$ of graphs such that the nodes of G_i are V_i and such that distances in G_i are same as in G. To keep these graphs sparse, we include exactly each edge (u, v) in graph G_{i+1} for which all shortest u-v-paths in G_i do not contain any node of V_{i+1} as inner node. The length of an edge (u, v) in G_i is the distance from u to v in G.

The *level i* of a node v, is the highest index i such that $v \in V_i$. The *multilevel overlay graph G'* is given by $G' := G[E_1 \cup \ldots \cup E_L]$. The query is a bidirectional search in G', where, outgoing from a node of level i, only edges in $E_i \cup \ldots \cup E_L$ are relaxed. Pseudocode of preprocessing phase and query are given as Algorithm 3.3 and Algorithm 3.4, an example as Figure 3.4.

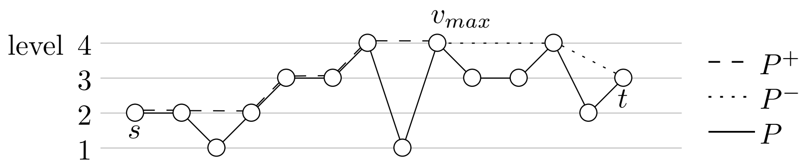
Algorithm 3.3: Preprocessing-Phase of the Multilevel Overlay Graph-Technique

```
input : graph $G = (V, E, \text{len})$, number of levels $L + 1$,
        sequence $V := V_0 \supseteq V_1 \supseteq \ldots \supseteq V_L$
output: multilevel overlay graph $G'$, level : $V \to \mathbb{Z}_{\geq 0}$

1 $G_0 \leftarrow G$

2 for $l = 1, \ldots, L$ do
3     $E_l \leftarrow \{(s, t) \in V_l \times V_l \mid \forall$shortest $s$-$t$-paths $(s, u_1, \ldots, u_k, t)$ in $G_{l-1}$
4                                   it is $u_1, \ldots, u_k \notin V_l\}$
5     $\text{len}_l \leftarrow$ function from $E_l$ to $\mathbb{R}_{\geq 0}$ such that $\text{len}_l(u, v) = \text{dist}_G(u, v)$
6     $G_l \leftarrow (V_i, E_i, \text{len}_i)$

7 $G' \leftarrow G[E_1 \cup \ldots \cup E_L]$

8 for $v \in V$ do
9     level($v$) $\leftarrow \max\{i \mid 0 \leq i \leq L$ and $v \in V_i\}$
```

Algorithm 3.4: Unidirectional Pruned Search of a Multilevel Overlay Graph-Query

```
input  : graph $G' = (V, E \cup E', \text{len})$, node $x \in V$, $\text{level} : V \to \mathbb{Z}_{\geq 0}$
output: distance label $d()$
1 for $v \in V$ do $d(v) \leftarrow \infty$                    /* Initialization Phase */
2 $d(x) \leftarrow 0$ ; $Q$.INSERT$(x, 0)$
3 while not $Q$.ISEMPTY do                                       /* Main Phase */
4     $v \leftarrow Q$.EXTRACTMIN     /* $v$ is now contained in the search space */
5     for $(v, w) \in E \cup E'$ with $\text{level}(w) \geq \text{level}(v)$ do
6         if $d(v) + len(v, w) < d(w)$ then
7             $d(w) \leftarrow d(v) + len(v, w)$
8             $Q$.INSERTORUPDATE$(w, d(w))$
```

Correctness. As we slightly altered the Multilevel Overlay Graph-technique, we proof the correctness of our variant. Let $G = (V, E, \text{len})$ be the input graph and $G' = (V, E \cup E', \text{len}')$ be the according multilevel overlay graph with level function $\text{level} : V \to \mathbb{Z}_{\geq 0}$. Given is a pair of nodes $s, t \in V$. As distances are equal in G and G' an s-t query can not compute a value smaller than $\text{dist}(s, t)$. This already proves the case $\text{dist}(s, t) = \infty$, in the following we assume $\text{dist}(s, t) < \infty$. We now construct two paths P^+ and P^- that are shortest-paths in G' and that are witnesses for the fact that an s-t-query computes a value of at most $\text{dist}(s, t)$. A visualization of P^+ and P^- is given in the following picture, the corresponding shortest s-t-path in G is denoted by P.



We use the notation $P^{\bowtie}(x, y)$ for the set of all nodes that lie on a shortest x-y-path. This includes x and y, see page 68 for a formal definition. Let v_{max} be an arbitrary node in $P^{\bowtie}(s, t)$ of maximum level. We construct P^+: We start with $P^+ = v_{\text{max}}$. We iteratively proceed as follows until $P^+ =: (v_1, \ldots, v_k)$ is such that $v_1 = s$: Let v_{next} be an arbitrary node in $P^{\bowtie}(s, v_1) \setminus v_1$ of maximum level and such that $\text{dist}(v_{\text{next}}, v_1)$ is minimal among all such nodes. Set $P^+ := (v_{\text{next}}, v_1, \ldots, v_k)$.

Let $P^+ = (v_1, \ldots, v_k)$ be the sequence resulting from that procedure. It is $\text{level}(v_i) \leq \text{level}(v_{i+1})$ for $i = 1, \ldots, k-1$ because of the maximality of v_{max}. It is $(v_i, v_{i+1}) \in E \cup E'$: If $(v_i, v_{i+1}) \in E$ nothing is to show. Let $(v_i, v_{i+1}) \notin E$. Let $d = \text{level}(v_i)$. It is $v_i \in G_d$ and $v_{i+1} \in G_d$. If there was a node $w \in G_d$ such that $\text{dist}(v_i, v_{i+1}) = \text{dist}(v_i, w) + \text{dist}(w, v_{i+1})$ the node v_i would not be the predecessor of v_{i+1} in P^+ as w would have been chosen instead (remember that edge lengths are strictly positive). Consequently, there is no such node w which implies that $(v_i, v_{i+1}) \in E_d \subseteq E'$. Hence, P^+ gets settled in the forward search starting at s.

We analogously construct P^-: We start with $P^- = v_{\text{max}}$. We iteratively proceed as follows until $P^+ =: (v_1, \ldots, v_k)$ is such that $v_k = t$: Let v_{next} be an arbitrary node in $v \in P^{\bowtie}(v_k, t) \setminus v_k$ of maximum level and such that $\text{dist}(v_k, v_{\text{next}})$ is minimal among all such nodes. Set $P^+ := (v_1, \ldots, v_k, v_{\text{next}})$.

Analogously, the sequence P^- gets settled in the backward search starting at t. From the construction of P^+ and P^- follows that $\text{len}(P^+) + \text{len}(P^-) = \text{dist}(s, t)$ which shows the correctness.

The Problem. We study the problem of preprocessing the Multilevel Overlay Graph-technique. We want to minimize the expected search space under the restriction that the size of the preprocessed data is bounded by a given parameter. This problem turns out to be NP-hard even when restricting to 2 levels. We denote by $\mathrm{OVL}(G, V_1)$ the set of *overlay edges* introduced to G when choosing the sequence V, V_1 as input to the preprocessing, i.e., $\mathrm{OVL}(G = (V, E, \mathrm{len}), V_1)$ equals the set

$$\{(s,t) \in V \times V \mid \forall \text{shortest } s\text{-}t\text{-paths } (s, u_1, \dots, u_k, t) \text{ in } G \text{ it is } u_1, \dots, u_k \notin V_1\} \setminus E.$$

Further, let $V_{V_1}(s, t, G)$ be the search space of an s-t-Multilevel Overlay Graph-query whose input (G', level) results from preprocessing G with sequence V, V_1.

Problem MinOVL. Given a graph $G = (V, E, \mathrm{len})$ and an integer $F \geq |E|$, choose $V_1 \subseteq V$, such that $|E \cup \mathrm{OVL}(G, V_1)| \leq F$ and $\sum_{s,t \in V} |V_{V_1}(s, t, G)|$ is minimal.

We observe that a feasible solution always must exist as we could choose $V_1 = V$.

Theorem 3. Problem MinOVL is NP-hard.

Proof. We make a reduction from Exact Cover by 3-Sets (X3C, page 11). Given an instance (U, C) of X3C with $|U| = 3q$, we construct an instance $(G = (V, E, \mathrm{len}), F = |E|)$ of MinOVL as follows, see Figure 3.5 for a visualization. The set V consists of a node b, one node c for each $c \in C$, one node u for each $u \in U$ and a set M_v of M additional nodes for each node v in $\{b\} \cup U$ where M will be specified later. We set $D := \bigcup_{v \in \{b\} \cup U} M_v$. For each $c \in C$, there is a directed edge (b, c). For each $u \in U$ and each $c \in C$, such that $u \in c$, there is a directed edge (c, u). Finally, for each $v \in \{b\} \cup U$ and each $w \in M_v$, there is an undirected edge $\{v, w\}$. All edges have uniform length 1. The transformation is polynomial as M will be polynomial in the input size Note that, as $F = |E|$, no new edges may be introduced to the overlay-graph, i.e., $\mathrm{OVL}(G, V_1)$ is empty.

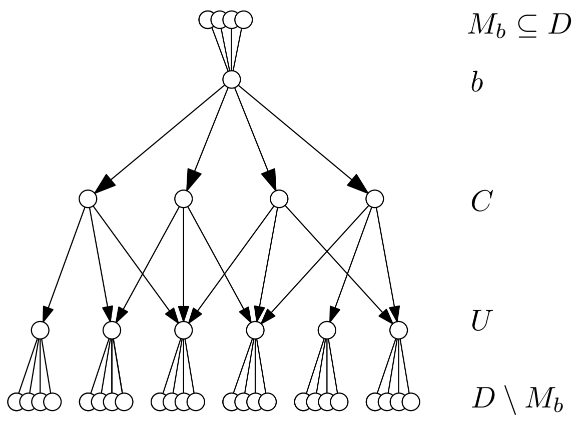


Figure 3.5: Graph G constructed from the X3C-instance $\{1, 2, 3\}, \{2, 3, 4\}, \{3, 4, 6\}, \{4, 5, 6\}$, some nodes in D are omitted.

Requisite. Throughout the remaining proof, we write $X := (|U| + |C| + 2)$ and set $M := 2^5 X^5 + 1$. We will see that, under this assumption, the maximum search space of an s-t-query is at most $2X$ when V_1 is chosen appropriately. We use the search space characterization given as Equation 3.1.

Claim I. For each V_1 with $\sum_{s,t\in V} |V_{V_1}(s,t)| \le 2X|V|^2$, we have $\{b\} \cup U \subseteq V_1$.
Assume to the contrary that $b \notin V_1$. Then

$$\begin{aligned}\sum_{s,t\in V} |V_{V_1}(s,t)| &= \sum_{z\in V} |V| \left(|V^+_{V_1}(z)| + |V^-_{V_1}(z)|\right) \ge \sum_{z\in M_b} |V| \left(|V^+_{V_1}(z)| + |V^-_{V_1}(z)|\right) \\ &\ge |V| \cdot M \cdot 2M \ge 2|V|M2^5X^5 > 2|V|^2X\end{aligned}$$

as, for each node $z \in M_b$, all nodes in M_b get settled from forward and backward search rooted at z. The same holds analogously for any node $u \in U$ not in V_1.

Claim II. Let V_1 be a solution with $\sum_{s,t\in V} |V_{V_1}(s,t)| \le 2X|V|^2$. Then, for each $u \in U$, there is at least one $c \in C$ with $u \in c$ and $c \in V_1$.

Recall that $\{b\} \cup U \subseteq V_1$. Assume, that there is a u with $\{c \in C \cap V_1 \mid (c,u) \in E\} = \emptyset$. Then, the set E_1 would contain the edge (u,b) which contradicts $F = |E|$.

Claim III. There is a number B polynomial in $|V|$ and M such that the following holds: There is a $V_1 \subseteq V$ with $\sum_{s,t\in V} |V_{V_1}(s,t)| \le \min\{B, 2X|V|^2\}$ if and only if (C,U) has an exact cover.

'*If*'. For an exact cover C^* of (C,U) let $V^* := C^* \cup U \cup \{b\}$. It is $\sum_{s,t\in V} |V_{V^*}(s,t)| \le 2X|V|^2$ as for each s-t-pair at most the root and the nodes in $\{b\} \cup C \cup U$ get settled by one direction of an s-t-query. Further, no new edges are introduced during the preprocessing. It is easy to verify the following list of search space sizes.

z	$V^+_{V^*}(z)$	$V^-_{V^*}(z)$
$\in M_b$	$2+q+\lvert U\rvert$	1
b	$1+q+\lvert U\rvert$	1
$\in C$	4	2
$\in U$	1	3
$\in D \setminus M_b$	1	4

Summing over all possible source-target pairs yields B:

$$B := M(3+q+|U|) + 2 + q + |U| + 6|C| + 4|U| + 5|M||U|$$

'*Only if*'. Let (C,U) be such that there is no exact cover for (C,U). Let $V' \subset V$ be such that $\sum_{s,t\in V} |V_{V'}(s,t)| \le 2X|V|^2$. Then V' has the structure as described in Claims *I* and *II*. It is easy to verify that the above tabular for the search-space sizes of V^* gives lower bounds for the search space sizes of V'. Further, as there is no exact cover, we know that $|V^+_{V'}(b)| > 1 + q + |U|$. Hence, $\sum_{s,t\in V} |V_{V'}(s,t)| > B$.

Summary. We have shown that the decision variant of problem MinOVL that asks if it is possible to reach a certain search-space size is NP-hard. Hence, also the optimization variant, i.e., problem MinOVL is NP-hard. □

A note on the Original Model. This chapter is based on the papers [BCK+10a, BCK+10b]. There, the model for the Multilevel Overlay Graph-query also includes a stopping criterion for the bidirectional search. We decided to relax the stopping criterion in this thesis. This way, we have a uniform description for each bidirectional technique and focus more on the aspects of the Multilevel Overlay Graph-approach that cause the main speedup. Further, we could strongly simplify the proof. However, the proof in [BCK+10a, BCK+10b] also shows that it is not problematic to also consider a stopping criterion.

3.5 ALT

Goal-Directed Search. Goal-Directed Search (also called A^*-search) is a variant of Dijkstra's algorithm which assigns a different priority to the nodes in the queue [HNR68]. This alternative priority shall guide the search more into the direction of the target. When applying Dijkstra's algorithm, the priority of node v in the queue equals the tentative distance $d(v)$. Goal-Directed Search adds a potential $\Pi_t(v)$ depending on the target t to the node's priority, i.e., the priority of v when applying Goal-Directed Search is $d(v) + \Pi_t(v)$.

An equivalent formulation is as follows. Given a graph $G = (V, E, \text{len})$ and the potential function $\Pi_t : V \to \mathbb{R}_{\geq 0}$, Goal-Directed Search on $G' = (V, E, \text{len})$ is Dijkstra's algorithm on $\overline{G} := (V, E, \overline{\text{len}})$ where

$$\overline{\text{len}}(u, v) = \text{len}(u, v) - \Pi_t(u) + \Pi_t(v)$$

for each edge $(u, v) \in E$. The length $\overline{\text{len}}(P)$ of an s-v-path $P = (s = v_1, \dots, v_{k+1} = v)$ in $\overline{G}$ is then

$$\begin{aligned} \overline{\text{len}}(P) &= \sum_{i=1}^{k} \overline{\text{len}}(v_i, v_{i+1}) = \sum_{i=1}^{k} \text{len}(v_i, v_{i+1}) - \Pi_t(v_i) + \Pi_t(v_{i+1}) \\ &= -\Pi_t(s) + \Pi_t(v) + \sum_{i=1}^{k} \text{len}(v_i, v_{i+1}) = -\Pi_t(s) + \Pi_t(v) + \text{len}(P). \end{aligned}$$

Hence both formulations are equal except that priorities in the queue differ by the constant $\Pi_t(s)$. Further, shortest s-t paths in $\overline{G}$ are also shortest paths in G' which shows correctness of the approach *if* the edge lengths $\overline{\text{len}}$ are positive. We assure this by using only *feasible* potentials: A potential is called feasible if $\overline{\text{len}}(u, v) \geq 0$ for every edge (u, v). One can show that, given two feasible potentials Π_t^1 and Π_t^2, the potential Π_t defined by $\Pi_t(v) := \max\{\Pi_t^1(v), \Pi_t^2(v)\}$ is also feasible.

ALT. The ALT-algorithm [GH05, GW05] is Goal-Directed Search with a special potential function. Initially, a set $L \subset V$ of 'landmarks' is chosen. For a landmark $l \in L$ we define

$$\Pi_t^{l+}(v) := \text{dist}(v, l) - \text{dist}(t, l) \tag{3.4}$$
$$\Pi_t^{l-}(v) := \text{dist}(l, t) - \text{dist}(l, v) \tag{3.5}$$

We work with graphs that are not strongly connected. Hence, infinite distances may occur. When computing landmark potentials, we use the conventions $\infty - \infty := 0$, $\infty - n := \infty$ and $n - \infty := 0$ for any $n \in \mathbb{R}_{\geq 0}$. A set L of landmarks defines a potential by taking the maximum over all single landmark potentials:

$$\Pi_t^L(v) := \max_{l \in L} \left\{ \Pi_t^{l+}(v), \Pi_t^{l-}(v), 0 \right\}.$$

Note that this potential is feasible and that $\Pi_t^L(t) = 0$. We denote by $V_L(s, t, G)$ and $V_\Pi(s, t, G)$ the search space of an s-t-ALT-query in graph G using landmarks L and potential Π, respectively. The preprocessing phase of the ALT-technique consists of choosing the set of landmarks L and precomputing the distances from and to all of these landmarks. This information is the input to the query. As the distances are determined as soon as the landmarks are chosen, we consider the set of landmarks L to be the only additional data handed over to the query-phase. Pseudocode for the ALT-query is given as Algorithm 3.5.

Algorithm 3.5: ALT-Query

```
input  : graph $G = (V, E, \mathrm{len})$, source $s \in V$, target $t \in V$, landmarks $L \subseteq V$
output: distance label $d(t)$
1 for $v \in V$ do $d(v) \leftarrow \infty$                    /* Initialization Phase */
2 $d(s) \leftarrow 0$ ; $Q$.insert$(s, 0)$
3 while not $Q$.isEmpty do                                       /* Main Phase */
4     $v \leftarrow Q$.extractMin      /* $v$ is now contained in the search space */
5     if $v = t$ then stop algorithm
6     for $(v, w) \in E$ do
7         if $d(v) + len(v, w) < d(w)$ then
8             $d(w) \leftarrow d(v) + len(v, w)$
9             $Q$.InsertOrUpdate$(w, d(w) + \Pi_t^L(w))$
```

Correctness for General Graphs. We are aware of a formal proof of correctness only for the case of strongly connected graphs with positive edge weights. As we work on arbitrary graphs with positive edge weights, we briefly adapt the proof of correctness of Dijkstra's algorithm for our means. We start by proving some technical lemmata.

Lemma 1. From $\mathrm{dist}(v, t) < \infty$ follows that $\Pi_t^L(v) < \infty$ for any set of landmarks L.

Proof. We show that from $\Pi_t^L(v) = \infty$ follows that $\mathrm{dist}(v, t) = \infty$. Let L be a set of landmarks. If $\Pi_t^L(v) = \infty$ there must be a landmark $l \in L$ such that $\Pi_t^{l+}(v) = \infty$ or $\Pi_t^{l-}(v) = \infty$. Let $\Pi_t^{l+}(v) = \infty$. This is equal to $\mathrm{dist}(v, l) = \infty$ and $\mathrm{dist}(t, l) < \infty$. With $\infty = \mathrm{dist}(v, l) \leq \mathrm{dist}(v, t) + \mathrm{dist}(t, l)$ we have that $\mathrm{dist}(v, t) = \infty$ as $\mathrm{dist}(t, l) < \infty$. Let $\Pi_t^{l-}(v) = \infty$. This is equal to $\mathrm{dist}(l, t) = \infty$ and $\mathrm{dist}(l, v) < \infty$. With $\infty = \mathrm{dist}(l, t) \leq \mathrm{dist}(l, v) + \mathrm{dist}(v, t)$ we have $\mathrm{dist}(v, t) = \infty$ as $\mathrm{dist}(l, v) < \infty$. □

We now generalize the notion of *feasibility* for arbitrary graphs.

Definition. We call a potential Π_t *feasible* if $\mathrm{len}(x, y) - \Pi_t(x) + \Pi_t(y) \geq 0$ for every edge $(x, y) \in E$ such that $\mathrm{dist}(y, t) < \infty$.

Note that the definition of *feasibility* is well-defined as, with Lemma 1, we have that $\Pi_t(x) < \infty$ and $\Pi_t(y) < \infty$.

Lemma 2. Let Π_t and Π'_t be feasible potentials. Then, the potential $\Pi_t^{\max}$ given by $\Pi_t^{\max}(v) = \max\{\Pi_t(v), \Pi'_t(v)\}$ is feasible.

Proof. Let $(x, y) \in E$ be such that $\mathrm{dist}(y, t) < \infty$. Then $\mathrm{len}(x, y) - \Pi_t(x) + \Pi_t(y) \geq 0$ and $\mathrm{len}(x, y) - \Pi'_t(x) + \Pi'_t(y) \geq 0$ implies that $\mathrm{len}(x, y) - \Pi_t(x) + \max\{\Pi_t(y), \Pi'_t(y)\} \geq 0$ and $\mathrm{len}(x, y) - \Pi'_t(x) + \max\{\Pi_t(y), \Pi'_t(y)\} \geq 0$ which yields $\mathrm{len}(x, y) - \max\{\Pi_t(x), \Pi'_t(x)\} + \max\{\Pi_t(y), \Pi'_t(y)\} \geq 0$. This was to show. □

Lemma 3. Let $L \subseteq V$ be a set of landmarks and Π^L be the potential induced by L. It is Π_t^L feasible for any target $t \in V$.

Proof. We show that, for any landmark $l \in V$, it is Π_t^{l+} and $\max\{\Pi_t^{l-}, 0\}$ feasible. The claim then follows with Lemma 2. Let $(x, y) \in E$ be such that $\mathrm{dist}(y, t) < \infty$.

$[\Pi_t^{l+}]$. Let $\mathrm{dist}(t,l)=\infty$. Then $\Pi_t^{l+}(x)=\Pi_t^{l+}(y)=0$ and $\mathrm{len}(x,y)-\Pi_t^{l+}(x)+\Pi_t^{l+}(y)\geq 0$ holds. Let $\mathrm{dist}(t,l)<\infty$. Because of $\mathrm{dist}(x,t)<\infty$ we have that $\mathrm{dist}(x,l)<\infty$ and $\mathrm{dist}(y,l)<\infty$. Hence, $\mathrm{len}(x,y)-\Pi_t^{l+}(x)+\Pi_t^{l+}(y)=\mathrm{len}(x,y)-\mathrm{dist}(x,l)+\mathrm{dist}(y,l)$. This is at least 0 because of $\mathrm{len}(x,y)+\mathrm{dist}(y,l)\geq\mathrm{dist}(x,l)$.

$[\max\{\Pi_t^{l-},0\}]$. Let Π_t' denote $\max\{\Pi_t^{l-},0\}$. Let $\mathrm{dist}(l,t)=\infty$. Because of $\mathrm{dist}(l,y)+\mathrm{dist}(y,t)\geq\mathrm{dist}(l,t)$ and $\mathrm{dist}(l,x)+\mathrm{dist}(x,t)\geq\mathrm{dist}(l,t)$ it is $\mathrm{dist}(l,x)=\infty$ and $\mathrm{dist}(l,y)=\infty$. Accordingly, $\Pi_t^{l-}(x)=\Pi_t^{l-}(y)=0$ and $\mathrm{len}(x,y)-\Pi_t'(x)+\Pi_t'(y)=\mathrm{len}(x,y)\geq 0$ holds. In the following let $\mathrm{dist}(l,t)<\infty$. If $\mathrm{dist}(l,x)=\infty$ then $\Pi_t^{l-}(x)=0$ and $\mathrm{len}(x,y)-\Pi_t'(x)+\Pi_t'(y)=\mathrm{len}(x,y)+\Pi_t'(y)\geq 0$. Hence, in the following let $\mathrm{dist}(l,x)<\infty$. This implies $\mathrm{dist}(l,y)<\infty$. It is $\mathrm{len}(x,y)-\Pi_t'(x)+\Pi_t'(y)=\mathrm{len}(x,y)-\max\{0,\Pi_t^{l-}(x)\}+\max\{\Pi_t^{l-}(y),0\}$. Hence, if $\Pi_t^{l-}(x)\leq 0$ it is $\mathrm{len}(x,y)-\Pi_t'(x)+\Pi_t'(y)\geq 0$. Now let $\Pi_t^{l-}(x)\geq 0$. Then $\mathrm{len}(x,y)-\Pi_t'(x)+\Pi_t'(y)\geq\mathrm{len}(x,y)-\Pi_t^{l-}(x)+\Pi_t^{l-}(y)=\mathrm{len}(x,y)+\mathrm{dist}(l,x)-\mathrm{dist}(l,y)$. This is at least zero because of $\mathrm{dist}(l,x)+\mathrm{len}(x,y)\geq\mathrm{dist}(l,y)$. □

Equipped with these preparatory lemmata we show the correctness of the ALT-query.

Theorem 4. The ALT-Query Algorithm 3.5 computes $d(t)=\mathrm{dist}(s,t)$ for any set of Landmarks L.

Proof. As $d(v)<\infty$ represents the length of a path in G it is $d(v)\geq\mathrm{dist}(s,v)$ for any $v\in V$. This already settles the case $\mathrm{dist}(s,t)=\infty$. In the remainder of the proof let $\mathrm{dist}(s,t)<\infty$. We show that, for each $v\in V$ with $\mathrm{dist}(v,t)<\infty$ it is $d(v)=dist(s,v)$ when v gets settled. This holds in particular for $v=t$.

We assume the contrary. Let u be the first node with $\mathrm{dist}(u,t)<\infty$ such that $\mathrm{dist}(s,u)<d(u)$ when u gets settled. We denote by S the set of nodes that get settled before u. It is $s\in S$. Let $P=(s=u_1,\ldots,u_k=u)$ be a shortest s-u-path. There must be a vertex in $P\setminus\{u\}$ that is not contained in S: Otherwise $u_{k-1}\in S$ and with $\mathrm{dist}(u_{k-1},t)<\infty$ we have that $d(u_{k-1})=\mathrm{dist}(s,u_{k-1})$ when u_{k-1} gets settled. The relaxation of edge (u_{k-1},u) gives the contradiction $d(u)=\mathrm{dist}(s,u)$ when u gets settled.

Let u_ℓ be the first vertex in P that is not contained in S. Then, when u gets settled, it is $\mathrm{dist}(s,u_\ell)=d(u_\ell)$ because $u_{\ell-1}$ gets settled before u. Further u_ℓ is in the queue at the time when u gets settled. As u_ℓ gets settled after u we have

$$\mathrm{dist}(s,u_\ell)+\Pi_t^L(u_\ell)\geq d(u)+\Pi_t^L(u)>\mathrm{dist}(s,u)+\Pi_t^L(u)\ .$$

As $\mathrm{dist}(u,t)<\infty$ we can apply Lemma 1 on any node in P. Hence, all potentials of nodes in P are smaller than infinity. This allows for the following transformation.

$$\begin{aligned}0 &> \mathrm{dist}(s,u_k)-\mathrm{dist}(s,u_\ell)-\Pi_t^L(u_\ell)+\Pi_t^L(u_k)\\ &= \sum_{i=\ell}^{k-1}\mathrm{len}(u_i,u_{i+1})+\sum_{i=\ell}^{k-1}\big(-\Pi_t^L(u_i)+\Pi_t^L(u_{i+1})\big)\end{aligned}$$

This is a contradiction to the feasibility of the potential, i.e., to $\mathrm{len}(x,y)-\Pi_t^L(x)+\Pi_t^L(y)\geq 0$ for any y with $\mathrm{dist}(y,t)<\infty$. Consequently, we have a contradiction to the assumption that there is an $u\in V$ with $\mathrm{dist}(u,t)<\infty$ such that $d(u)>\mathrm{dist}(s,u)$ when u gets settled. In particular, this implies that $d(t)=\mathrm{dist}(s,t)$ when t gets settled. □

Some helpful Lemmata. The next lemma shows that, when using landmarks, the potential of a node is a lower bound of its distance to the target.

Lemma 4. Let $L\subseteq V$ be a set of landmarks and $t\in V$ be a vertex. Then, for any vertex v, it is $\Pi_t^L(v)\leq\mathrm{dist}(v,t)$.

Proof. We consider Π_t^{l+} and Π_t^{l-} separately for any landmark $l \in L$. The lemma then follows with $\Pi_t^L(v) := \max_{l \in L}\{\Pi_t^{l+}(v), \Pi_t^{l-}(v), 0\}$ and distances always being non-negative.

$[\Pi_t^{l+}(v)]$. Let $\text{dist}(t,l) = \infty$. Then $\Pi_t^{l+}(v) = 0$ and the claim follows. Now let $\text{dist}(t,l) < \infty$. If $\text{dist}(v,l) = \infty$ then $\Pi_t^{l+}(v) = \infty$. With Lemma 1 follows $\text{dist}(s,v) = \infty$ and the claim follows. Now let $\text{dist}(v,l) < \infty$. Then we have $\Pi_t^{l+}(v) = \text{dist}(v,l) - \text{dist}(t,l) \le \text{dist}(v,t)$ because of $\text{dist}(v,l) \le \text{dist}(v,t) + \text{dist}(t,l)$.

$[\Pi_t^{l-}(v)]$. Let $\text{dist}(l,v) = \infty$. Then $\Pi_t^{l-}(v) = 0$ and the claim follows. Now let $\text{dist}(l,v) < \infty$. If $\text{dist}(l,t) = \infty$ it is $\Pi_t^{l-}(v) = \infty$. With Lemma 1 follows $\text{dist}(s,v) = \infty$ and the claim follows. Now let $\text{dist}(l,t) < \infty$. Then we have $\Pi_t^{l-} = \text{dist}(l,t) - \text{dist}(l,v) \le \text{dist}(v,t)$ because of $\text{dist}(l,t) \le \text{dist}(l,v) + \text{dist}(v,t)$. □

The next lemma shows that nodes that can be reached from a node with infinite potential also have infinite potential.

Lemma 5. Let $t \in V$ be a target, $(u,v) \in E$ and $L \subseteq V$. If $\Pi_t^L(u) = \infty$ then $\Pi_t^L(v) = \infty$.

Proof. If $\Pi_t^L(u) = \infty$ then there is a landmark $l \in L$ such that $\Pi_t^{l+}(u) = \infty$ or $\Pi_t^{l-}(u) = \infty$. Let $\Pi_t^{l+}(u) = \infty$. Then it is $\text{dist}(u,l) = \infty$ and $\text{dist}(t,l) < \infty$. This implies $\text{dist}(v,l) = \infty$ Hence, $\Pi_t^{l+}(v) = \infty$. Let $\Pi_t^{l-}(u) = \infty$. Then $\text{dist}(l,t) = \infty$ and $\text{dist}(l,u) < \infty$. This implies that $\text{dist}(l,v) < \infty$. Hence, $\Pi_t^{l-}(v) = \infty$. □

The next lemma helps to bound the search space.

Lemma 6. Let $s,t \in V$ be such that $\text{dist}(s,t) < \infty$. Let $w \in V$ be an additional node. If $\text{dist}(s,w) + \Pi_t^L(w) > \text{dist}(s,t)$ then $w \notin \mathcal{V}_L(s,t)$

Proof. Consider an arbitrary shortest s-t-path $P = (s = v_1, \dots, v_k = t)$. Until $t = v_k$ is settled, there is always one node $v_i \in P$ in the queue, such that $\text{dist}(s,v_i) = d(v_i)$. Initially, this is $v_i = s$. Further, if a node $v_i \in P \setminus \{t\}$ is removed from Q, the edge (v_i, v_{i+1}) is relaxed and the same holds for v_{i+1}. With Lemma 4 we know that $\text{dist}(s,v_i) + \Pi_t^L(v_i) \le \text{dist}(s,v_i) + \text{dist}(v_i,t) = \text{dist}(s,t)$. Hence, until t gets settled, there is always one node with priority at most $\text{dist}(s,t)$ in the queue. As $\text{dist}(s,w) + \Pi_t^L(w)$ is a lower bound for the priority of node w and $\text{dist}(s,w) + \Pi_t^L(w) > \text{dist}(s,t)$, node w cannot get settled before t. □

The Problem. The problem MinALT is that of assigning a given number of landmarks to a graph (and thus using only a given amount of preprocessing space), such that the expected size of the search space is minimal.

Problem MinALT. Given a directed graph $G = (V, E, \text{len})$ and an integer r, find a set $L \subseteq V$ with $|L| = r$ such that $\sum_{s,t \in V} V_L(s,t) := \sum_{s,t \in V} V_L(s,t,G)$ is minimal.

Theorem 5. Problem MinALT is NP-hard.

Proof. We make a reduction from 3-MinimumCover (see page 11). Given is an instance (C, U, k) of 3-MinimumCover. We say, a *set* $c \in C$ *covers* an *element* $u \in U$ if $u \in c$. W.l.o.g we may assume that for each $u \in U$ there is a set $c \in C$ that covers u. Further, as a preparatory step, we may assume that $|c| = 3$ for each set c and that, for each element u, there is a set c that does not cover u. To assure this, we first remove each element that is contained in every set, as such elements do not affect the solvability of the instance. In case each set c of the remaining instance has size of at most 2, we can solve the problem in polynomial time [GJ79]. Otherwise, we transform the resulting instance $(C', U', k' = k)$

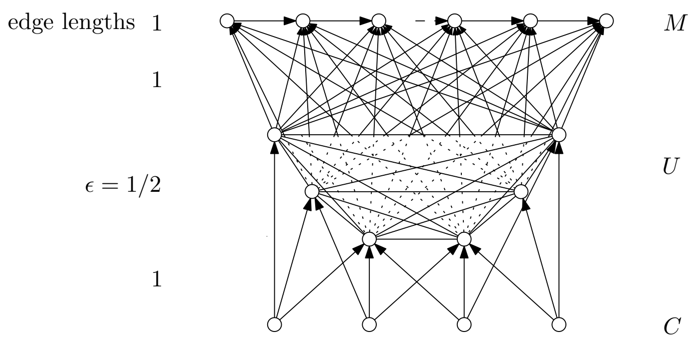


Figure 3.6: Graph G constructed from the 3MinimumCover-instance $\{1,2,3\},\{2,3,4\},\{3,4,6\},\{4,5,6\}$.

to a new instance (C^*, U^*, k^*) by setting $k^* := k' + 1$, $U^* := U' \cup \{x,y,z\}$ where x, y, z are new elements and

$$\begin{aligned} C^* := \{x,y,z\} \quad &\cup \quad \{c \mid c \in C', |c| = 3\} \\ &\cup \quad \{c \cup \{x\} \mid c \in C', |c| = 2\} \\ &\cup \quad \{c \cup \{x,y\} \mid c \in C', |c| = 1\} \end{aligned}$$

This instance fulfills our claims as each set $c^* \in C^*$ has cardinality 3 by construction and, for $u \in U$, the set $\{x,y,z\} \in C^*$ does not cover u. Further, for $u \in \{x,y,z\}$, there is a set in C^* that does not cover u as there is a set in C' of cardinality 3. Finally, (C^*, U^*, k^*) is a yes-instance if and only if (C, U, k) is a yes-instance: The only way to cover z is to incorporate the set $\{x,y,z\}$ to the cover. Then, x, y and z are covered and the problem immediately reduces to instance (C', U', k').

Given the 3-MinimumCover-instance (C, U, k) of the form described above, we construct an instance $(G = (V, E, \text{len}), r = k+1)$ of MinALT as follows, see Figure 3.6 for a visualization. We introduce a set $M = \{m_1, \ldots, m_{|M|}\}$ of nodes, with $|M|$ to be specified later and assign $V := C \cup U \cup M$. There is an edge (c, u) with length 1 for each $u \in c$. Moreover, there is an edge (u_i, u_j) with length $\epsilon := 0.5$ for each pair of elements $u_i \in U$, $u_j \in U$, $u_i \neq u_j$, an edge (u, m) with length 1 for each pair $u \in U, m \in M$ and an edge (m_i, m_{i+1}) with length 1 for each $i = 1, \ldots, |M| - 1$. For breaking ties in the queue, nodes labels are such that the nodes have the order $c_1 \leq \ldots \leq c_{|C|} \leq u_1 \leq \ldots \leq u_{|U|} \leq m_1 \leq \ldots \leq m_{|M|}$. The number of landmarks r is set to be $k+1$. The transformation is polynomial as we later choose $|M|$ polynomial.

All following claims are assembled in Claim VII. This can also be used as an outline for the proof. Let L be a set of landmarks. Then

$$\sum_{s,t \in V} V_L(s,t) = \underbrace{\sum_{s \in V, t \in C} V_L(s,t) + \sum_{s \in M, t \in U \cup M} V_L(s,t)}_{=:\alpha} + \underbrace{\sum_{s \in U \cup C, t \in U} V_L(s,t)}_{\leq \beta} + \underbrace{\sum_{s \in U \cup C, t \in M} V_L(s,t)}_{=:\gamma(L)}$$

Claim I. The value of α is independent of the choice of L and can be computed in polynomial time.

For the cases $s \in V, t \in C$ and $s \in M, t \in U$, the target t is not reachable from the source s (unless $s = t$). Therefore, in these cases, the search space is exactly the number of nodes that are reachable from s (or 1 if $s = t$). If $s, t \in M$, either the target is again not reachable from the source or the search space is a direct path to the target.

Claim II. It holds $0 \leq \sum_{s \in U \cup C, t \in U} V_L(s,t) \leq \beta := |U||C|(1+|U|) + |U|^3$.
Let Π be an arbitrary potential induced by landmarks and $t \in U$. As Π is induced by landmarks, we have $\Pi \geq 0$. In case $s \in C$, it holds

$$\operatorname{dist}(s,t) + \Pi_t(t) = \operatorname{dist}(s,t) \leq 1 + \epsilon < 2 = \operatorname{dist}(s,m) \leq \operatorname{dist}(s,m) + \Pi_t(m)$$

for $m \in M$. With Lemma 6 follows, that for each of the $|C||U|$ s-t-queries in $C \times U$, nodes in M are not settled. This leaves us with at most $1 + |U|$ settled nodes for such queries. In case $s \in U$, it holds

$$\operatorname{dist}(s,t) + \Pi_t(t) = \operatorname{dist}(s,t) \leq \epsilon < 1 = \operatorname{dist}(s,m) \leq \operatorname{dist}(s,m) + \Pi_t(m)$$

for $m \in M$. With Lemma 6 follows, that for each of the $|U|^2$ s-t-pairs in $U \times U$, nodes in M are not settled. This leaves us with at most $|U|$ settled nodes for such queries.

Definition. We call a set L of landmarks *set-covering* if $m_1 \in L$ and for each $u \in U$, either u or a node $c \in C$ with $u \in c$ is contained in L.

Claim III. Let $L \subseteq V$ be set-covering. Then $\gamma(L) := \sum_{s \in U \cup C, t \in M} V_L(s,t) \leq |M|(5|C| + 2|U|)$.
Remember the given order on V to break ties in the queue. Let $s \in U \cup C$ and $t \in M$. It is $V_L(s,t) \cap M = \{t\}$: The value $\operatorname{dist}(s,m)$ is equal for all $m \in M$. Because of the landmark m_1 we have $\Pi_t^L(m) > 0$ for $m < t, m \in M$. As $\Pi_t(t) = 0$ we know that nodes $m \in M$ with $m < t$ do not get settled. Because of $\Pi_t^L(v) \geq 0$ for each node v and the breaking-ties rule, nodes $m \in M$ with $m > t$ do not get settled.

As L is set-covering, we have for any $u \in U$ a landmark $l \in L$ such that

$$\Pi_t^L(u) \geq \operatorname{dist}(l,t) - \operatorname{dist}(l,u) = \begin{cases} 1 - 0 = 1 & , u = l \\ 2 - 1 = 1 & , \exists l \in C \cap L \text{ s.t. } u \in l. \end{cases}$$

Hence, for $s, u \in U$ with $s \neq u$, we have

$$\operatorname{dist}(s,u) + \Pi_t^L(u) \geq \epsilon + 1 > 1 = \operatorname{dist}(s,t) + \Pi_t^L(t)$$

which implies with Lemma 6 that $V_L(s,t) = \{s,t\}$. Accordingly, for $s \in C$, we have that for each $u \in U$ with $u \notin s$ it is

$$\operatorname{dist}(s,u) + \Pi_t^L(t) \geq 1 + \epsilon + 1 > 2 = \operatorname{dist}(s,t)$$

which with Lemma 6 implies that $V_L(s,t) \subseteq \{s,t\} \cup \{u \in U \mid s \text{ covers } u\}$. Summing over all possible source-target pairs shows the claim.

Claim IV. Let $s \in C, t \in M$ and Π_t be an arbitrary potential induced by landmarks. Then $|V_{\Pi_t}(s,t)| \geq 5$ and $V_{\Pi_t}(s,t) \supseteq \{s,t\} \cup \{u \in U \mid s \text{ covers } u\}$.

It is $\Pi_t(t) = 0$. With Lemma 4 we have that $\Pi_t(u) \leq 1 \leq \operatorname{dist}(u,t)$ for any $u \in U$. Consequently, for any $u \in s$ it is $\operatorname{dist}(s,u) + \Pi_t(u) \leq 2 = \operatorname{dist}(s,t) + \Pi_t(t)$. With the order of breaking ties in the queue, we have that each $u \in s$ is contained in $V_{\Pi_t}(s,t)$ which shows the claim.

Claim V. Let L be a set of landmarks with $M \cap L = \emptyset$. Then $\sum_{s \in C \cup U, t \in M} V_L(s,t) \geq (|C| + |U|)|M|(|M|+1)/2$.
If there is no landmark in M, then all nodes in M have the same potential due to symmetry reasons. Hence, before a target $t \in M$ is settled from a source $s \notin M$, all nodes $m \in M$

with $m < t$ are settled first. The claim follows by summing over all possible sources and targets.

Claim VI. Let L be a set of landmarks such that there is an $x \in U$ that is not a landmark and not covered by a landmark. Then $\gamma(L) := \sum_{s \in U \cup C, t \in M} V_L(s,t) \geq |M|(5|C| + 2|U|) + |M| - |L|$.

Let t be in M. Let s be in U, then $|V_L(s,t)| \geq 2$ as source and target differ. Let s be in C, by Claim IV we know that $|V_L(s,t)| \geq 5$. This already shows $\gamma(L) \geq |M|(5|C| + 2|U|)$. Because of the preparatory step, we know that there is a set $c_x \in C$ with $x \notin c_x$. Furthermore, there are at least $|M| - |L|$ nodes in M that are no landmarks. Let $t_x \in M$ denote such a node. We now show that $x \in V_L(c_x, t_x)$ and hence $V_L(c_x, t_x) \supseteq \{c_x, t_x, x\} \cup \{u \in U \mid s \text{ covers } u\}$: For a landmark l, it is

$$\Pi^l_{t_x}(x) = \max\{\Pi^{l+}_{t_x}, \Pi^{l-}_{t_x}, 0\} = \begin{cases} \max\{0, 2-(1+\epsilon)\} = 1-\epsilon & , l \in C \quad (\text{as } x \notin l) \\ \max\{0, 1-\epsilon\} = 1-\epsilon & , l \in U \quad (\text{as } u \neq l) \\ \max\{0,0\} = 0 & , l \in M, l < t_x \quad (\text{as } t_x \notin L) \\ \max\{0,0\} = 0 & , l \in M, l > t_x \quad (\text{as } t_x \notin L) \end{cases}$$

Consequently, we have

$$\text{dist}(c_x, x) + \Pi^L_{t_x}(x) \leq 1 + \epsilon + 1 - \epsilon = \text{dist}(c_x, t_x) + \Pi^L_{t_x}(t_x).$$

It is x with $d(x) = \text{dist}(c_x, x)$ in the queue as soon as the first node in U is settled. As the node order is such that $x \leq t_x$ we have that x gets settled before t_x. Summarizingly, there are at least $|M| - |L|$ nodes $t_x \in M$ that are no landmarks. For each of them, the node x gets settled in the c_x-t_x query. The claim now follows by summing over all possible source-target pairs in $(U \cup C) \times M$.

Claim VII. Let $|M| := \max\{\beta + r, 2(\beta + 5|C| + 2|U| + 1)\} + 1$. Then, there exists a set L of landmarks such that $\sum_{s,t \in V} V_L(s,t) \leq \alpha + \beta + |M|(5|C| + 2|U|) =: q$ if and only if (C,U) has a cover of size at most k.

To prove that claim we assemble all preceding claims. '*If*'. Let C' be a set cover of (C,U) of size k. Then $C' \cup \{m_1\}$ is a set-covering set of landmarks. With Claim I, II and III we know that $\sum_{s,t \in V} V_L(s,t) \leq q$. '*Only if*'. Let L be such that $\sum_{s,t \in V} V_L(s,t) \leq q$. It is $|L \setminus M| \leq k$: By construction we have $|L| = k+1$. If $L \cap M$ would be the empty set, then we have with Claim V that

$$\sum_{s,t \in V} V(s,t) \geq \alpha + (|C| + |U|)|M|(|M|+1)/2 > \alpha + \beta + |M|(5|C| + 2|U|) = q$$

which contradicts the assumption $\sum_{s,t \in V} V(s,t) \leq q$. Hence $|L \setminus M| \leq k$. Let the set C'' contain each landmark in $L \cap C$ and, for each $u \in L \cap U$, an arbitrary $c \in C$ with $u \in c$. It is $|C''| \leq |L \setminus M|$. Further, C'' is a set cover of (C,U). Assume the contrary. Then, with Claim VI

$$\sum_{s,t \in V} V(s,t) \geq \alpha + |M|(5|C| + 2|U|) + |M| - |L| > \alpha + \beta + |M|(5|C| + 2|U|) = q$$

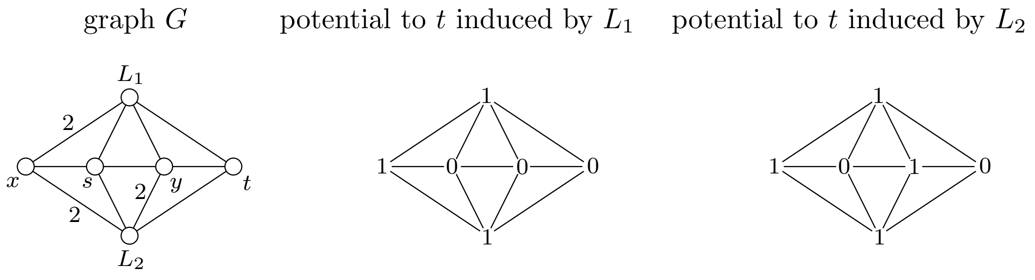
which contradicts the assumption $\sum_{s,t \in V} V(s,t) \leq q$.

Summary. We have shown that the decision variant of problem MinALT that asks whether it is possible to reach a certain search-space size is NP-hard. Hence, also the optimization variant, i.e., problem MinALT is NP-hard. □

A Note on the Original Proof. This chapter is based on the papers [BCK+10a, BCK+10b]. There, the proof of Theorem 5 makes use of the following lemma, which is taken out of [GH05]. Remember that, when extracting nodes from the queue, ties are broken by a predefined order on the nodes.

Lemma 7 (Theorem 4.1 in [GH05], wrong). *Given arbitrary nodes s, t and feasible potential functions Π and Π' with $\Pi(t) = \Pi'(t) = 0$ and $\Pi(v) \le \Pi'(v)$ for any vertex v. Then $V_{\Pi'}(s,t) \subseteq V_{\Pi}(s,t)$.*

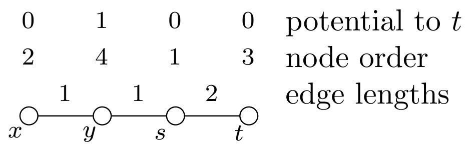
However, we later observed that this lemma is not correct. The following graph gives a counterexample:


graph G
potential to t induced by L_1
potential to t induced by L_2
L_1
L_2
x
s
y
t
2
2
2
1
0
0
0
1
1
0
1
0
1
1

Edge lengths are 1 unless stated otherwise. Nodes are ordered such that $t < x < y < L_1 < L_2$. Ties are broken due to that order when extracting nodes from the queue. The ALT-search space of an s-t-query is $\{s, y, t\}$ when using landmark L_1 and $\{s, x, y, t\}$ when using landmark L_2. However, landmark L_2 induces a *stronger potential*, i.e., $\Pi_{L_2}(v) \ge \Pi_{L_1}(v)$ for any node v in the graph. The underlying mistake is, that the search space $V_L(s,t)$ of an ALT-query can not generally be expressed as

$$\{v \in V \quad | \quad \mathrm{dist}(s,v) + \Pi_t^L(v) < \mathrm{dist}(s,t) \text{ or } \\ \mathrm{dist}(s,v) + \Pi_t^L(v) = \mathrm{dist}(s,t) \text{ and } v < t\}$$

which can be seen in the following example that, for simplicity, uses a feasible potential, not induced by a set of landmarks.



A Critical View on our Model. The model for the ALT-algorithm is, in several aspects, weaker than the other models in this chapter. First, the node order is not arbitrary: A *special* node order for breaking ties in the queue is specified within the reduction. Second, in contrast to the other techniques, the size of the preprocessed data directly influences the query time: More preprocessed landmarks require a longer time for computing the potentials. Hence, we can claim a close relationship between search space size and actual query time only for a constant number of landmarks. Finally, we have to note that our model does not exclude nodes with infinite potential from the search space which could be done because of Lemma 1. Including this could improve the runtime for graphs that are not strongly connected.

An Intuition for the ALT-search space. The general intuition behind A^*-search is to guide the search more into the direction of the target. We refine this intuition for ALT. We ignore the actual choice of the node ordering that is applied to break ties when extracting nodes from the queue and consider a worst-case ordering. Effects of this decision take place only at the 'border' of the search space. We now consider an s-t-query with landmarks L. The search space

$$V_d = \{v \in V \mid \operatorname{dist}(s,v) \leq \operatorname{dist}(s,t)\}$$

of Dijkstra's algorithm can be seen as a graph-theoretic circle. For each landmark $l \in L$, the potential $P_t^l(v)$ induced by l consists of an *elliptic* part

$$\Pi_t^{l+}(v) = \operatorname{dist}(v,l) - \operatorname{dist}(t,l)$$

and an *hyperbolic part*

$$\Pi_t^{l-}(v) = \operatorname{dist}(l,t) - \operatorname{dist}(l,v) \ .$$

For each part, we transform the equation

$$V^L(s,t) \subseteq \{v \in V \mid \operatorname{dist}(s,v) + \Pi^L(v) \leq \operatorname{dist}(s,t)\}$$

given by Lemma 6. The elliptic part $\Pi_t^{l+}(v)$ defines the graph-theoretic ellipse

$$V_{l^+}(s,t) = \{v \in V \mid \operatorname{dist}(s,v) + \operatorname{dist}(v,l) \leq \operatorname{dist}(s,t) + \operatorname{dist}(t,l)\}$$

with focuses s and l and node t at the boundary. The hyperbolic part $\Pi_t^{l-}(v)$ defines the graph-theoretic hyperbola

$$V_{l^-}(s,t) = \{v \in V \mid \operatorname{dist}(s,v) - \operatorname{dist}(l,v) \leq \operatorname{dist}(s,t) - \operatorname{dist}(l,t)\}$$

with focuses s and l and node t at the border. The search space of the ALT s-t-query is at most the intersection of the circle V_d and all ellipses $V_{l^+}(s,t)$ and hyperbolas $V_{l^-}(s,t)$ defined by the single landmarks $l \in L$. See Figure 3.7 for a visualization.

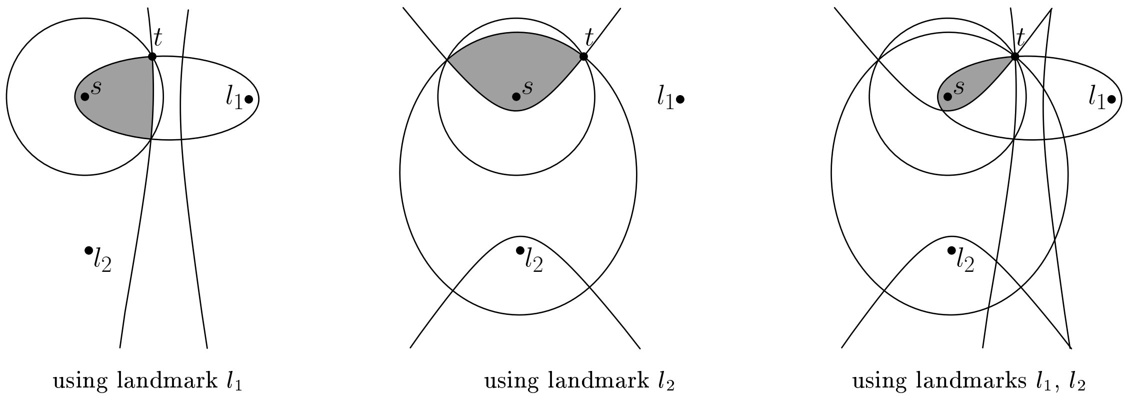


Figure 3.7: Schematic visualization of the ALT search space of an s-t-query.

3.6 Arc-Flags

Given the underlying graph $G = (V, E, \text{len})$, this unidirectional approach [Lau04, KMS05, MSS+05, MSS+06, Sch06a, HKMS06, HKMS09] works as follows: At first V is divided into k cells $\mathcal{V} = (V_1, V_2, \ldots, V_k)$ such that $V = V_1 \dot\cup V_2 \dot\cup \ldots \dot\cup V_k$. We call $\mathcal{V}$ a k-partition of G. For a node w, we write $\mathcal{V}(w) = V_i$ if $w \in V_i$.

The main idea of the approach is to identify, during the preprocessing phase, some combinations of edges (u, v) and cells V_i, such that we do not have to consider edge (u, v) in a search with target in V_i. Afterwards, this information is attached to the edges as an *Arc-Flag-Vector* $\mathcal{F}_{(\cdot,\cdot)}(\cdot)$: It is $\mathcal{F}_{(u,v)}(V_i) = \text{false}$ if we already know that we do not have to follow edge (u, v) in a search with target in cell V_i and true otherwise. The Arc-Flags s-t-query is the variant of Dijkstra's algorithm which only relaxes edges (u, v) for which $\mathcal{F}_{(u,v)}(\mathcal{V}(t))$ is true. The pseudocode is given as Algorithm 3.6. We denote by $\mathcal{V}_{\mathcal{F}}(s, t, G)$ the search space of an ArcFlags$s$-t-query on graph G when using Arc-Flag-Vector $\mathcal{F}$.

Algorithm 3.6: Arc-Flags Query

```
input : graph $G = (V, E, \text{len})$, source $s \in V$, target $t \in V$
        Arc-Flag-Vector $\mathcal{F}$, partition $\mathcal{V}$
output: distance label $d(t)$
1 for $v \in V$ do $d(v) \leftarrow \infty$                 /* Initialization Phase */
2 $d(s) \leftarrow 0$ ; $Q$.insert$(s, 0)$
3 while not $Q$.isEmpty do                                /* Main Phase */
4     $v \leftarrow Q$.extractMin     /* $v$ is now contained in the search space */
5     if $v = t$ then stop algorithm
6     for $(v, w) \in E$ with $\mathcal{F}_{(v,w)}(\mathcal{V}(t)) = \text{true}$ do
7         if $d(v) + len(v, w) < d(w)$ then
8             $d(w) \leftarrow d(v) + len(v, w)$
9             $Q$.InsertOrUpdate$(w, d(w))$
```

Obviously, wrong choices of $\mathcal{F}$ may lead to incorrect queries. An Arc-Flag-Vector $\mathcal{F}$ is *correct* if the following holds: For each pair $s, t \in V$ there is a shortest s-t path P such that, for every edge $e \in P$, it is $\mathcal{F}_e(\mathcal{V}(t)) = \text{true}$. Throughout this section we assume that all Arc-Flag-Vectors are correct. Figure 3.8 shows that there can be several correct Arc-Flag-Vectors. There are two degrees of freedom within the Arc-Flags preprocessing phase:

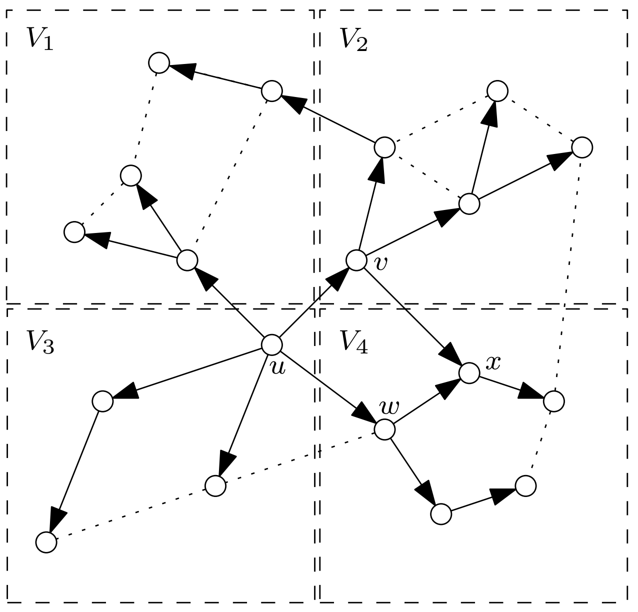
(1) The choice of a k-partition.

(2) Given a k-partition $\mathcal{V}$, the choice of an Arc-Flag-Vector $\mathcal{F}$.

Given a k-partition $\mathcal{V}$, a straightforward way to compute a correct Arc-Flag-Vector is to compute *all* shortest paths in the graph and set $\mathcal{F}$ as follows:

$$\mathcal{F}_{(u,v)}(V_i) := \begin{cases} \text{true} & , \exists \text{ shortest path starting with } (u, v) \text{ and ending with a node in } V_i \\ \text{false} & , \text{ otherwise} \end{cases}$$

We denote this choice of the Arc-Flag-Vector by $\mathcal{F}^{\text{all}(\mathcal{V},G)}$. In Section 3.6.1 we work on the problem of how to choose the k-partition $\mathcal{V}$ such that the average ArcFlags-search space is minimized when using Arc-Flag-Vector $\mathcal{F}^{\text{all}(\mathcal{V},G)}$.



Graph G and shortest-paths subgraph T with root u. Edges that are not contained in T are drawn dashed.

Computing the Arc-Flag-Vector straightforwardly, we gain

- (true, true, false, true) for (u, v)
- (false, false, false, true) for (u, w)

However, we stay correct if we set the vector to

- (true, true, false, false) for (u, v)

as shortest u-x-paths can also use the node w.

Figure 3.8: Example for the Arc-Flags-preprocessing.

Section 3.6.2 focuses on the second degree of freedom. We are already given a k-partition and the problem is to compute an Arc-Flag-Vector that minimizes the average ArcFlags-search space. In Section 3.6.3 we additionally consider shortcuts (see Section 3.3.2 and Chapter 4). This models an enhancement of the Arc-Flags-technique that is used within the Sharc-algorithm [BD08, BD09, Del09, BDGW10]. We assume that a k-partition is already given and we have the following additional degree of freedom:

(3) Given a k-partition $\mathcal{V}$, the choice of a set of shortcuts.

In the presence of shortcuts, it would not make much sense to use the vector $\mathcal{F}^{\mathrm{all}(\mathcal{V},G)}$. Remember that we uniquely identify each node with an integer. A shortest path $(v_1, v_2, \ldots, v_\ell)$ is called *canonical* if it is edge-minimal among all shortest v_1-v_ℓ-paths and if $(v_1, \ldots, v_\ell)$ is lexicographically minimal among all edge-minimal shortest v_1-v_ℓ-paths. When working with shortcuts, we compute the Arc-Flag-Vector by

$$\mathcal{F}_{(u,v)}(V_i) := \begin{cases} \text{true} & , \exists \text{ canonical shortest path starting with } (u,v) \\ & \quad \text{and ending with a node in } V_i \\ \text{false} & , \text{ otherwise} \end{cases}$$

and denote this choice of the Arc-Flag-Vector by $\mathcal{F}^{\mathrm{can}(\mathcal{V},G)}$. Accordingly, given a k-partition $\mathcal{V}$ we study the problem of how to insert a set of shortcuts $\mathcal{S}$ to the graph such that the average ArcFlags-search space is minimized when working with Arc-Flag-Vector $\mathcal{F}^{\mathrm{can}(\mathcal{V},G[\mathcal{S}])}$. Note that our results also hold when relaxing the definition of canonical shortest paths to an arbitrary choice that favors edge-minimal shortest paths.

For all problems, the preprocessing size is bounded. Hence, the number of cells and the number of shortcuts is an input parameter of the problems.

3.6.1 Main Technique: Choosing a Partition

In this section we show the NP-hardness of finding a k-partition $\mathcal{V}$ such that the average ArcFlags-search space is minimized when using vector $\mathcal{F}^{\text{all}(\mathcal{V},G)}$.

Problem ArcFlags. Given a graph $G = (V, E, \text{len})$ and an integer k, find a k-partition $\mathcal{V} = \{V_1, \ldots, V_k\}$ of V such that $\sum_{s,t \in V} V_{\mathcal{V}}(s,t,G) := \sum_{s,t \in V} \mathcal{V}_{\mathcal{F}^{\text{all}(\mathcal{V},G)}}(s,t,G)$ is minimal.

We use the following technical lemmata.

Problem (P). Given numbers $L, m \in \mathbb{Z}_{>0}$, choose $c_1, c_2, \ldots, c_m \in \mathbb{Z}_{\geq 0}$ such that

$$\sum_{i=1}^{m} c_i(c_i + 1)/2$$

is minimized under the constraint $c_1 + c_2 + \ldots + c_m = Lm$.

Lemma 8. The only optimal solution of Problem (P) is $c_1^* = c_2^* = \ldots = c_m^* = L$ with objective value $D = mL(L+1)/2$. For any other feasible solution $c'_1, c'_2, \ldots, c'_m$ it is $\sum_{i=1}^{m} c'_i(c'_i + 1)/2 \geq D + 1$.

Proof. Given an arbitrary m-tuple $c' = (c'_1, c'_2, \ldots, c'_m)$ with $c'_1 + c'_2 + \ldots + c'_m = Lm$, we can construct c' by starting with the m-tuple $c = (c_1 = L, c_2 = L, \ldots, c_m = L)$ and iteratively proceeding as follows. At each step we choose some i and j fullfilling $c_i > c'_i$, $c_j < c'_j$ and set $c_i := c_i - 1$ and $c_j := c_j + 1$ until no such i and j exist anymore.

We define $\Delta^+ := \sum_{i|c'_i > c_i} c'_i - c_i$ and $\Delta^- := \sum_{i|c'_i < c_i} c_i - c'_i$. Throughout the construction we have the invariant $\Delta^+ = \Delta^-$. With each step, Δ^+ and Δ^- monotonically decrease by 1. Hence, $\Delta^+ = \Delta^- = 0$ and c' is constructed when the algorithm terminates (which is guaranteed due to monotonicity Δ^+).

Further, throughout the construction we have a second invariant: It is $c_i \leq c_j$ for each pair i, j with $c_i > c'_i$ and $c_j < c'_j$. Finally, when performing a step, the objective value increases by

$$\big[\underbrace{(c_i - 1)c_i + (c_j + 1)(c_j + 2)}_{\text{values after step}} - \underbrace{c_i(c_i + 1) - c_j(c_j + 1)}_{\text{values before step}}\big]/2 = c_j - c_i + 1 \geq 1$$

which proves the lemma. $\square$

Problem (P'). Given are a set A of $3m$ elements, a bound $B \in \mathbb{Z}_{>0}$ and a size w_a for each $a \in A$ such that $B/4 < w_a < B/2$ and such that $\sum_{a \in A} w_a = mB$. Partition A into m disjoint sets $A_1, A_2, \ldots, A_m$ such that $\sum_{i=1}^{m} \left(|A_i| \cdot \sum_{a \in A_i} w_a\right)$ is minimal.

Lemma 9. An optimal solution of Problem (P') has to fullfill $|A_i| = 3$ for each $i = 1, \ldots, m$ and has objective value $D = 3\sum_{a \in A} w_a = 3mB$. Further, for any other partition of A the objective value is at least $D + 1$.

Proof. We can construct an arbitrary partition $A^* = (A_1^*, A_2^*, \ldots A_m^*)$ analogous to the proof of Lemma 8: There is a starting partition $A' = (A'_1, A'_2, \ldots A'_m)$ with $|A'_i| = 3$ such that we can construct A^* out of A' by iteratively moving one element from a set A_r with $|A_r| \leq 3$ to a set A_s with $|A_s| \geq 3$. We gain A' as follows. First, we only consider the sizes of the cells A'_i. Accordingly, we choose A' such that each A'_i contains exactly three dummy elements. Then we proceed as in the proof of Lemma 8 such that at the end $|A'_i| = |A_i^*|$

holds. Afterwards we know all necessary move-operations. Hence, we re-consider the initial choice of A' and exchange the dummy elements with elements in A such that after the procedure it is $A'_i = A^*_i$.

Let $A_r = \{1, \dots, k\}$ and $A_s = \{k+1, \dots, k+\ell\}$. When moving element k to A_s the objective function increases by

$$\underbrace{(k-1)\sum_{i=1}^{k-1} w_i + (\ell+1)\sum_{i=k}^{k+\ell} w_i}_{\text{new values}} - \underbrace{k\sum_{i=1}^{k} w_i - \ell \sum_{i=k+1}^{k+\ell} w_i}_{\text{old values}}.$$

This simplifies to

$$(\ell+1-k)w_k + \sum_{i=k+1}^{k+\ell} w_i - \sum_{i=1}^{k-1} w_i > B/4 + 3 \cdot B/4 - 2B/2 = 0$$

because of $k \leq 3$ and $\ell \geq 3$ and the bounds on the w_i. The claim follows with the objective function always being integer-valued. □

Theorem 6. Problem ArcFlags is NP-hard.

The following proof often uses Equation 3.2 given in Section 3.2.

Proof. We make a reduction from the strongly NP-complete problem 3-Partition (see page 11). Given is a 3-Partition-instance $(A, \{w_a | \ a \in A\})$ with $|A| = 3m$ for an $m \in \mathbb{Z}_{>0}$ and $\sum_{a \in A} w_a = Bm$ for a $B \in \mathbb{Z}_{>0}$. Remember that $B/4 < w_a < B/2$ for each element $a \in A$. Hence, if an instance is a yes-instance, the corresponding solution has to consist of sets of cardinality 3. We construct an ArcFlags-instance $(G = (V, E, \text{len}), m+1)$ as follows, a visualization is given as Figure 3.9.

Initially, G is the empty graph. We introduce a directed cycle of $|Z|$ nodes to G. More exactly, we add the subgraph (Z, U, len') with

$$\begin{aligned} Z &= \{z_1, \dots, z_{|Z|}\} \\ U &= \{(z_i, z_{i+1}) \mid 1 \leq i \leq |Z|-1\} \cup \{(z_{|Z|}, z_1)\}) \\ \text{len}'(u,v) &= 1/(|Z|+1) \text{ for each } (u,v) \in U. \end{aligned}$$

The cardinality $|Z|$ will be specified later. We further introduce the set A to V and, for each $a \in A$, a set W_a of w_a nodes. We denote by W the set $\bigcup_{a \in A} W_a$. There is a directed edge (z, w) of length 1 for each $z \in Z$ and $w \in W$ and a directed edge (w, a) of length 1 for each $a \in A$ and $w \in W_a$. The transformation is polynomial as we later choose $|Z|$ to be polynomial in the input size.

The proof outlines as follows. We first show that, for $|Z|$ big enough, there is an optimal k-partition such that Z is a separate cell. We then decompose the objective function and show that queries from Z to the remaining graph dominate the search space size (as we consider queries within Z as fixed because Z is one cell). With this knowledge and the preparatory technical lemmata we establish a connection between solutions of $(A, \{w_a | \ a \in A\})$ and $(G, m+1)$.

Claim. Let $Z > 6|A \cup W|^3$. Then there is an optimal solution $\mathcal{V} = (V_1, \dots, V_{m+1})$ of ArcFlags-instance $(G, m+1)$ such that $V_{m+1} = Z$.

We assume $Z > 6|A \cup W|^3$. In Step 1 we give an upper bound for the search space size of any partition $\mathcal{V}^*$ for which Z is exactly the union of some cells in $\mathcal{V}^*$. Then, we show

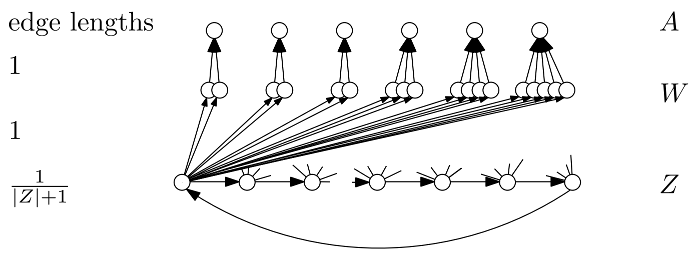


Figure 3.9: Graph G constructed from the 3-Partition-instance 2,2,2,3,4,5.

that such a partition can be transformed into a partition $\mathcal{V}^{**}$ of the claimed form without worsening the search space size. In Step 2 we show that each partition $\mathcal{V}'$ not considered in Step 1, i.e., each partition for which there is a cell containing nodes in Z and $V \setminus Z$ cannot be optimal: Then, the search space size is greater than the bound computed in Step 1.
[Step 1.] Let $\mathcal{V}^* = (V_1^*, \dots, V_{m+1}^*)$ be an $m+1$ partition such that there is a $J \subseteq \{1, 2, \dots, m+1\}$ with $Z = \bigcup_{j \in J} V_j^*$. Then, the vector $\mathcal{F}^{\text{all}(\mathcal{V}^*, G)}$ is such that queries with

- $s, t \in Z$ settle exactly the s-t-path in Z.
  Hence, $\sum_{s,t \in Z} V_{\mathcal{V}^*}(s,t) = |Z|^2(|Z|+1)/2$ as we have $|Z|$ different sources with $|Z|(|Z|+1)/2$ being the sum of the search spaces sizes over all $|Z|$ targets per source.
- $s \in Z$, $t \in A \cup W$ settle at most $\{s\} \cup A \cup W$.
  Hence, $\sum_{s \in Z, t \in A \cup W} V_{\mathcal{V}^*}(s,t) \leq |Z| \cdot (|A \cup W| + 1)^2 \leq |Z| \cdot |A \cup W|^3$.
- $s, t \in A \cup W$ settle at most $A \cup W$.
  Hence, $\sum_{s,t \in A \cup W} V_{\mathcal{V}^*}(s,t) \leq |A \cup W|^3$.
- $s \in A \cup W$, $t \in Z$ settle exactly $\{s\}$.
  Hence, $\sum_{s \in A \cup W, t \in Z} V_{\mathcal{V}^*}(s,t) = |Z| \cdot |A \cup W|$.

Assembling this yields

$$\sum_{s,t \in V} V_{\mathcal{V}^*}(s,t) = \underbrace{\sum_{s,t \in Z} V_{\mathcal{V}^*}(s,t)}_{=|Z|^2(|Z|+1)/2} + \underbrace{\sum_{s \in Z, t \in A \cup W} V_{\mathcal{V}^*}(s,t)}_{\substack{\leq |Z| \cdot (|A \cup W|+1)^2 \\ \leq |Z| \cdot |A \cup W|^3}} + \underbrace{\sum_{s,t \in A \cup W} V_{\mathcal{V}^*}(s,t)}_{\leq |A \cup W|^3} + \underbrace{\sum_{s \in A \cup W, t \in Z} V_{\mathcal{V}^*}(s,t)}_{=|Z| \cdot |A \cup W|}$$

W.l.o.g let $\max(J)$ be $(m+1)$. We now transform $\mathcal{V}^*$ to the partition $\mathcal{V}^{**}$ of the desired form and same search spaces. Let $\mathcal{V}^{**} = (V_1^{**}, \dots, V_{m+1}^{**})$ be the m-partition that results from unioning all cells V_i^* with $i \in J$, i.e., such that

$$V_i^{**} = \begin{cases} V_i^* & , i \notin J \\ \emptyset & , i \in J \setminus \{m+1\} \\ \bigcup_{j \in J} V_j^* & , i = m+1 \ . \end{cases}$$

Then $\sum_{s,t \in V} V_{\mathcal{V}^*}(s,t) = \sum_{s,t \in V} V_{\mathcal{V}^{**}}(s,t)$ which can be seen by the above decomposition of $\sum_{s,t \in V} V_{\mathcal{V}^*}(s,t)$: Search spaces with target not in V_{m+1}^{**} do not change as the according flags do not change. Search spaces with target in V_{m+1}^{**} still are optimal.
[Step 2.] On the other hand, let $\mathcal{V}' = (V_1', \dots, V_{m+1}')$ be an $(m+1)$-partition such that there are vertices $w \in A \cup W$ and $z \in Z$ with $w \in V_i'$ and $z' \in V_i'$ for some i. Then

- $\sum_{s,t \in Z} V_{\mathcal{V}'}(s,t) \geq |Z|^2(|Z|+1)/2$ as for each pair $s,t \in Z$ the path from s to t has to be settled.

- $\sum_{s \in Z} V_{\mathcal{V}'}(s,w) \geq |Z|(|Z|+1)/2$ as for each $z \in Z$ it is $\text{dist}(z,z') < 1 \leq dist(z,w)$. Hence, for each source $z \in Z$ and the target w, all nodes on the path from z to z' get settled.

Assembling this yields

$$\sum_{s,t \in V} V_{\mathcal{V}'}(s,t) \geq \sum_{s,t \in Z} V_{\mathcal{V}'}(s,t) + \sum_{s \in Z} V_{\mathcal{V}'}(s,w) \geq \frac{|Z|^2(|Z|+1)}{2} + \frac{|Z|(|Z|+1)}{2} \; .$$

We now compare the upper bound for the sum of the search space sizes of partition $\mathcal{V}^*$ with the lower bound for $\mathcal{V}'$. With the assumption $Z > 6|A \cup W|^3$ we have

$$\begin{aligned} \sum_{s,t \in V} V_{\mathcal{V}^*}(s,t) &\leq \frac{|Z|^2(|Z|+1)}{2} + |Z| \cdot |A \cup W|^3 + |A \cup W|^3 + |Z| \cdot |A \cup W| \\ &< \frac{|Z|^2(|Z|+1)}{2} + \frac{|Z|(|Z|+1)}{2} \leq \sum_{s,t \in V} V_{\mathcal{V}'}(s,t) \end{aligned}$$

because of

$$|Z| \cdot |A \cup W|^3 + |A \cup W|^3 + |Z| \cdot |A \cup W| < |Z|(|Z|+1)/2 \; .$$

Thus, we have $\sum_{s,t \in V} V_{\mathcal{V}^*}(s,t) < \sum_{s,t \in V} V_{\mathcal{V}'}(s,t)$ and $\mathcal{V}'$ cannot be optimal. This finishes Step 2 and proves the claim.

Requisite. In the remainder we assume $Z := 6|A \cup W|^3 + 1$. Further $\mathcal{V}^* = (V_1^*, \ldots, V_{m+1}^*)$ denotes an optimal solution of ArcFlags-instance $(G, m+1)$ such that $V_{m+1}^* = Z$. We decompose the objective function as follows (with some abuse of notation).

$$\sum_{s,t \in V} V_{\mathcal{V}^*}(s,t) = \Big(\underbrace{\sum_{s,t \notin Z}}_{\leq |W \cup A|^3} + \underbrace{\sum_{s,t \in Z} + \sum_{s \notin Z, t \in Z}}_{=:\alpha} + \underbrace{\sum_{s \in Z, t \in A}}_{=:\beta(\mathcal{V}^*)} + \underbrace{\sum_{s \in Z, t \in W}}_{=:\gamma(\mathcal{V}^*)} \Big) V_{\mathcal{V}^*}(s,t)$$

The value of α is independent of $\mathcal{V}^*$ as it equals $|Z|^2(|Z|+1)/2 + |W \cup A||Z|$. We write

$$\begin{aligned} A_i &:= A \cap V_i^* \\ \overline{W}_i &:= W \cap V_i^* \\ H_i &:= \{w \in W \mid w \in W_a \text{ and } a \in A_i\}. \end{aligned}$$

Note that $|H_i|$ is exactly the weight of all elements in A_i. Remember that there is an arbitrary order $<$ on the set of nodes which is used to break ties when extracting nodes from the queue. A query using partition $\mathcal{V}^*$ with source $z \in Z$ and

- target $a \in A_i$ settles z, a, each node $a' \in A_i$ with $a' < a$ and each node in W which are in the same cell as a or have a successor in the same cell as a (i.e., each node in $\overline{W}_i \cup H_i$).

- target $w \in \overline{W}_i$ settles *at least* z, w and each node $w' \in \overline{W}_i$ with $w' < w$.

This leads to

$$\begin{aligned}\beta(\mathcal{V}^*) = \sum_{s\in Z, t\in A} V_{\mathcal{V}^*}(s,t) &= |Z| \sum_{i=1}^{m} \big(|A_i|(|A_i|+1)/2 + |A_i|(|\overline{W}_i \cup H_i|) + 1\big) \\ \gamma(\mathcal{V}^*) = \sum_{s\in Z, t\in W} V_{\mathcal{V}^*}(s,t) &\geq |Z| \sum_{i=1}^{m} \big(|\overline{W}_i|(|\overline{W}_i|+1)/2 + 1\big)\end{aligned}$$

Further, $\gamma(\mathcal{V}^*) = |Z| \sum_{i=1}^{m} \big(|\overline{W}_i|(|\overline{W}_i|+1)/2+1\big)$ in case $\overline{W_i} = H_i$ for all i.

Claim. Let

$$\begin{aligned}\beta^* &:= |Z| \sum_{i=1}^{m} (3(3+1)/2 + 3B + 1) \\ \gamma^* &:= |Z| \sum_{i=1}^{m} (B(B+1)/2 + 1)\end{aligned}$$

Then there is a partition V^* such that

$$\sum_{s,t\in V} V_{\mathcal{V}^*}(s,t) \leq |W \cup A|^3 + \alpha + \beta^* + \gamma^*$$

if, and only if, the 3-Partition-instance $(A, \{w_a |\ a \in A\})$ is a yes-instance.

'*If*'. Let $\overline{A}_1, \overline{A}_2, \ldots, \overline{A}_m$ be a 3-partition of $(A, \{w_a |\ a \in A\})$. We partition G such that for each $a \in \overline{A}_i$ it holds $a \in V_i^*$ and $W_a \subseteq V_i^*$. Then, for each i, $\overline{W}_i = H_i$, $|W_i| = B$ and $|A_i| = 3$. This implies $\sum_{s,t\in V} V_{\mathcal{V}^*}(s,t) \leq |W \cup A|^3 + \alpha + \beta^* + \gamma^*$.

'*Only if*'. On the other hand, let $\sum_{s,t\in V} V_{\mathcal{V}^*}(s,t) \leq |W \cup A|^3 + \alpha + \beta^* + \gamma^*$. We show that $A_1, \ldots, A_m$ is a 3-partition of A. We apply

- Lemma 8 on $\sum_{i=1}^{m} |A_i|(|A_i|+1)/2$ with $\sum_{i=1}^{m} |A_i| = 3m$.
- Lemma 8 on $\sum_{i=1}^{m} |\overline{W}_i|(|\overline{W}_i|+1)/2$ with $\sum_{i=1}^{m} |\overline{W}_i| = Bm$
- Lemma 9 on $\sum_{i=1}^{m} |A_i||H_i|$ with $\sum_{i=1}^{m} |H_i| = mB$. This works as $|H_i| = \sum_{a\in A_i} w_a$.

By optimizing these terms separately we know that

$$\sum_{s,t\in V} V_{\mathcal{V}^*}(s,t) \geq \alpha + \beta^* + \gamma^* \ .$$

Further it is $|A_i| = 3$, $|\overline{W}_i| = B$ and $\overline{W}_i \subseteq H_i$ for all i, since otherwise

$$\sum_{s,t\in V} V_{\mathcal{V}^*}(s,t) \geq \alpha + \beta^* + \gamma^* + Z > |W \cup A|^3 + \alpha + \beta^* + \gamma^*.$$

From $\overline{W}_i \subseteq H_i$ follows $\overline{W}_i = H_i$ as both $\overline{W}_1, \ldots, \overline{W}_m$ and $H_1, \ldots, H_m$ partition W. Hence $|H_i| = B$ for all $i = 1, \ldots m$ and $A_1, \ldots A_m$ is a 3-Partition of A.

Summary. We have shown that we can solve problem 3-Partition in polynomial time with an algorithm that uses an oracle that gives an optimal solution of problem ArcFlags in constant time. This shows NP-hardness of problem ArcFlags. □

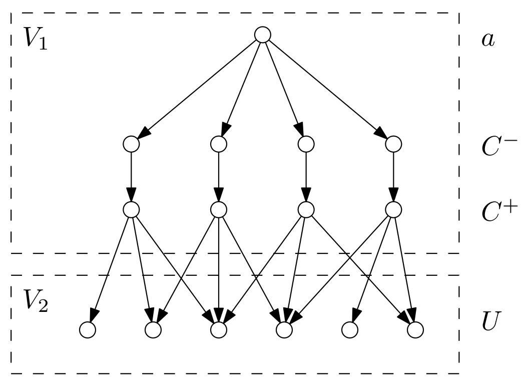


Figure 3.10: Graph G constructed from the X3C-instance $\{1,2,3\},\{2,3,4\},\{3,4,6\},\{4,5,6\}$.

3.6.2 Search-Space Minimal Arc-Flags.

This problem models a special aspect of Arc-Flags. We assume that the partition $\mathcal{V}$ is already given. Consider the vector $\mathcal{F} := \mathcal{F}^{\text{all}(\mathcal{V},G)}$. In case shortest paths are not unique, the situation may occur that one can improve the ArcFlags-search space by changing some values in $\mathcal{F}$ from true to false without violating the correctness of $\mathcal{F}$. See Figure 3.8 for an example.

We may assume that, for each edge (u,v) with $\mathcal{F}_{(u,v)}(V_i) =$ true, there is at least one shortest path starting with (u,v) that leads to cell V_i. The problem MinFlags is that of finding an Arc-Flag-Vector $\mathcal{F}$, such that the resulting average search space of an Arc-Flags-query is minimized.

Problem MinFlags. Given a graph $G = (V, E, \text{len})$ and a k-partition $\mathcal{V} = (V_1, \dots, V_k)$ of G, find a correct Arc-Flag-Vector $\mathcal{F}$ such that $\sum_{s,t \in V} V_{\mathcal{F}}(s,t) := \sum_{s,t\in V} V_{\mathcal{F}}(s,t,G)$ is minimal.

Theorem 7. Problem MinFlags is NP-hard (even for directed acyclic graphs).

Proof. We make a reduction from Exact Cover by 3-Sets (X3C, page 11). Let (U, C) be an instance of X3C with $|U| = 3q$. W.l.o.g. we may assume $\bigcup_{c\in C} = U$. We construct an instance $(G = (V, E, \text{len}), \mathcal{V} = \{V_1, V_2\})$ of MinFlags as follows, see Figure 3.10 for a visualization: The set V consists of two nodes c^- and c^+ for each $c \in C$, one node u for each $u \in U$ and one additional node a. The partition is given by $V_2 = U$, $V_1 = V \setminus V_2$. The graph G contains edges (a, c^-) and (c^-, c^+) for each $c \in C$, and an edge (c^+, u) if $u \in c$. All edges have equal length, the transformation is polynomial. The objective function can be decomposed into

$$\sum_{s,t\in V} V_{\mathcal{F}}(s,t) = \underbrace{\sum_{s\in V\setminus\{a\}, t\in V} V_{\mathcal{F}}(s,t) + \sum_{t\in V\setminus U} V_{\mathcal{F}}(a,t)}_{\alpha} + \sum_{t\in U} V_{\mathcal{F}}(a,t) \ .$$

Claim. The value of α is independent of $\mathcal{F}$ and can be computed in polynomial time.

Shortest paths that start with a node $u \neq a$ are unique. Therefore, an Arc-Flag-Vector for G is quite fixed: $\mathcal{F}_{(v,w)}(V_1)$ is true if and only if $w \notin V_2$. Further $\mathcal{F}_{(v,w)}(V_2)$ is true if $v \neq a$. The remaining degree of freedom is to decide for arbitrary c^- if $\mathcal{F}(a, c^-)(V_2)$ is true. For each node $s \neq a$, queries starting from s are not affected by the actual choice of $\mathcal{F}$. Further, queries for which $s, t \in V_1$ are also not influenced by the choice of $\mathcal{F}$ as the flags for cell V_1 are fixed.

Claim. There is an integer B such that (U, C) contains an exact cover if, and only if, there is an Arc-Flag-Vector $\mathcal{F}$ with $\sum_{s,t\in V} V_{\mathcal{F}}(s,t) \leq B$.

We call an arc-flag assignment an *exact cover of* G if, for each $u \in U$, the value $\mathcal{F}_{(a,c^-)}(V_2)$ is true for exactly one $c \in C$ with $u \in c$. Obviously an exact cover of G induces one of (U, C) and vice versa. Let $\mathcal{F}^*$ be an Arc-Flag-Vector that is an exact cover on G. Then

$$\beta^* := \sum_{t\in U} V_{\mathcal{F}^*}(a,t) = \underbrace{|U|}_{\#\text{targets}} \Big(\underbrace{1}_{a} + \underbrace{2q}_{\#\text{nodes in } C^- \cup C^+} \Big) + \underbrace{|U|(|U|+1)/2}_{\text{overall sum nodes in } U}$$

holds. The term $|U|(|U|+1)/2$ derives from the fact that the nodes in U are settled in some arbitrary but fixed order and before settling u all nodes v with $v < u$ get settled. We set $B := \alpha + \beta^*$. On the other hand, let $\mathcal{F}$ be an arbitrary Arc-Flag-Vector with $\sum_{s,t\in V} V_{\mathcal{F}}(s,t) \leq B$. It is

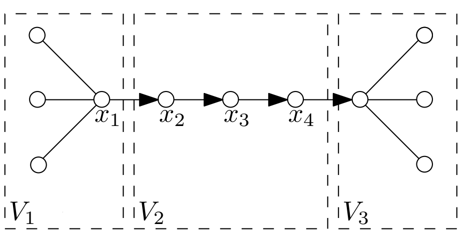
$$\sum_{t\in U} V_{\mathcal{F}}(a,t) = \underbrace{|U| \cdot 1}_{a} + 2|U|\#\{(a,c^-) \mid \mathcal{F}_{(a,c^-)}(V_2) = \text{true}\} + \underbrace{|U|(|U|+1)/2}_{\text{nodes in } U}.$$

From $\sum_{t\in U} V_{\mathcal{F}}(a,t) \leq \beta^*$ follows that $\#\{(a,c^-) \mid \mathcal{F}_{(a,c^-)}(V_2) = \text{true}\} \leq q$. As each set in C contains exactly 3 elements we need at least q sets to cover all $3q$ elements in U. The vector $\mathcal{F}$ has to cover each element in U to ensure correctness and hence it is $\#\{(a,c^-) \mid \mathcal{F}_{(a,c^-)}(V_2) = \text{true}\} = q$. This implies that $\mathcal{F}$ is an exact cover.

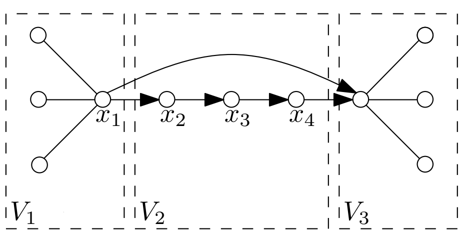
Summary. We have shown that the decision variant of problem MINFLAGS that asks if it is possible to reach a certain search-space size is NP-hard. Hence, also the optimization variant, i.e., problem MINFLAGS is NP-hard. □

3.6.3 External Shortcuts for Arc-Flags.

This problem models an enhancement of ARCFLAGS that is used within the SHARC-algorithm. We are already given a graph $G = (V, E, \text{len})$, a k-partition $\mathcal{V}$ and an integer ℓ. A *shortcut* is an edge (u, v) that is added to G for which $\text{len}(u, v) = \text{dist}(u, v)$, see Section 3.2 for a proper definition. The notation $G[E']$ denotes the graph G with the set E' of shortcuts added. Shortcuts can be used to alter the flags of some edges from true to false. The next picture gives an example.



flags for edges (x_i, x_{i+1}) V_1: false
V_2: true
V_3: true



flags for edges (x_i, x_{i+1}) V_1: false
V_2: true
V_3: false

We are allowed to add a set $\mathcal{S}$ of ℓ shortcuts to G, afterwards the vector $\mathcal{F}$ is computed by only considering canonical shortest paths, i.e., we use Arc-Flag-Vector $\mathcal{F}^{\text{can}(\mathcal{V},G[\mathcal{S}])}$ as input for the query. W.l.o.g. we do not insert shortcuts that are already present in the graph. Again, our aim is to minimize the average search space of the resulting ARC-FLAGS-query.

Problem ExtShortcutsArcFlags. Given a graph $G = (V, E, \text{len})$, a k-partition $\mathcal{V} = (V_1, \ldots, V_k)$ of V and a positive integer ℓ, find a shortcut assignment $\mathcal{S}$ with $|\mathcal{S}| \leq \ell$, such that $\sum_{s,t \in V} \mathcal{V}_\mathcal{S}(s,t) := \sum_{s,t\in V} \mathcal{V}_{\mathcal{F}^{\text{can}}(\mathcal{V},G[\mathcal{S}])}(s,t,G[\mathcal{S}])$ is minimal.

Theorem 8. Problem ExtShortcutsArcFlags is NP-hard (even for directed acyclic graphs).

For proving this theorem we require the following simple technical result.

Lemma 10. Given is a graph $G = (V, E, \text{len})$, a root $s \in V$ and a shortcut assignment $\mathcal{S}$. Then, Dijkstra's algorithm on G with root s settles nodes in the same order as Dijkstra's algorithm on $G[\mathcal{S}]$ with root s.

Proof. We make induction on the order in which nodes are settled in G. The initial step holds as s always is the first node settled. Now let the induction hypothesis hold for the first k settled nodes. Let u be the $k+1$th node settled in G. Assume that the $k+1$th node settled in $G[\mathcal{S}]$ is $w \neq u$. We have $\text{dist}(s,u) \leq \text{dist}(s,w)$ as u is settled in G before w. We have $\text{dist}(s,u) \geq \text{dist}(s,w)$ as u is settled in $G[\mathcal{S}]$ after w. Hence, $\text{dist}(s,u) = \text{dist}(s,w)$. Further, all remaining nodes v with $\text{dist}(s,v) = \text{dist}(s,u) = \text{dist}(s,w)$ are contained with $d(v) = \text{dist}(s,v)$ in the queue when u is settled in G and when w is settled in $G[\mathcal{S}]$. This holds in particular for u and w. This implies that $u < w$ and $w < u$ which is a contradiction. Hence, the assumption $w \neq u$ was wrong and the induction step is shown. □

Note that this lemma only holds as edge weights are strictly positive.

Proof (of Theorem 8). We make a reduction from Exact Cover by 3-Sets (X3C, page 11). Let (U, C) be an instance of X3C with $|U| = 3q$. W.l.o.g. we may assume $\bigcup_{c\in C} = U$. We construct an instance $(G = (V, E, \text{len}), \ell = q)$ of ExtShortcutsArcFlags as follows, see Figure 3.11 for a visualization. Initially, G is the empty graph. Then, we insert a path (h_1, h_2, a) into G. For each $u \in U$ we insert an edge (u, h_1) into G. For each $c \in C$ we insert a path (c, d_c^1, d_c^2, a) and M edges $(m_c^1, c), \ldots, (m_c^M, c)$ into G with M to be specified later. We denote by $\widetilde{M}$ the set $\{m_c^i \mid c \in C, 1 \leq i \leq M\}$ and by D_i the set $\{d_c^i \mid c \in C\}$. Finally, there is an edge $(u, c) \in U \times C$ if $u \in c$. The edge lengths are adjusted such that each path in G is a shortest path, i.e., all edge lengths are 1 except the length of edge (h_2, a) which equals 2. The partition $\mathcal{V} = (V_1, V_2)$ is given by $V_2 = \{a\}, V_1 = V \setminus V_2$. The transformation is polynomial as we later choose M to be polynomial in the input size. Let $\mathcal{S}$ be a shortcut assignment. It is

$$\sum_{s,t\in V} V_\mathcal{S}(s,t) = \underbrace{\sum_{s\in V, t\in V\setminus\{a\}} V_\mathcal{S}(s,t)}_{\alpha} + \underbrace{\sum_{s\in V\setminus\widetilde{M}} V_\mathcal{S}(s,a)}_{\leq\beta} + \underbrace{\sum_{s\in\widetilde{M}} V_\mathcal{S}(s,a)}_{\gamma(\mathcal{S})}$$

We first show that the value α is independent of the applied shortcut assignment and compute a value for the upper bound beta. We will see that, for large M, the value $\gamma(\mathcal{S})$ dominates the average search space size. Further, we show that an optimal shortcut assignment is contained in $C \times \{a\}$. With these tools we establish a relationship between an exact cover for (C, U) and a reasonable good shortcut assignment for G.

Claim. The value of $\alpha := \sum_{s\in V, t\in V\setminus\{a\}} V_\mathcal{S}(s,t)$ is independent of $\mathcal{S}$ and can be computed in polynomial time.

Each edge (v, w) in the graph induced by $V \setminus \{a\}$ is the only path from v to w. This does not change due to a shortcut insertion. Therefore $\mathcal{V}_{(v,w)}(V_1)$ is true if, and only if,

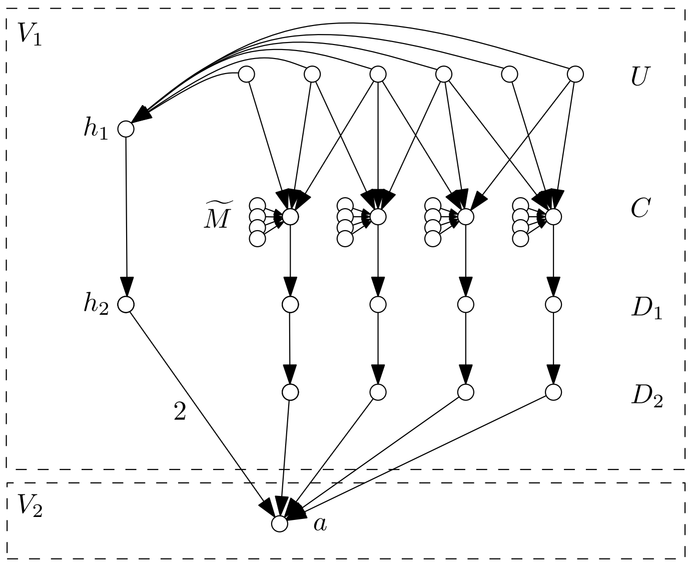


Figure 3.11: Graph G constructed from the X3C-instance $\{1,2,3\},\{2,3,4\},\{3,4,6\},\{4,5,6\}$.

$w \neq a$. Hence, as all flags concerning cell V_1 of edges in E are fixed. The Arc-Flags query is Dijkstra's algorithm on the subgraph for which the flags of the corresponding target cell are true. Additional shortcuts do not influence the search space of Dijkstra's algorithm, as shown in Lemma 10. Consequently, searches with target in V_1 do not depend on $\mathcal{S}$.

Claim. It is $\sum_{s \in V \setminus \widetilde{M}} \mathcal{V}_{\mathcal{S}}(s,a) \leq \beta := |V \setminus \widetilde{M}|^2$.

This holds as the corresponding part of the search space consists of $|V \setminus \widetilde{M}|$ different queries. For each query, at most the entire subgraph reachable from the source is settled. This subgraph is fully contained in $V \setminus \widetilde{M}$.

Definition. Note that β is independent of M. We now fix $M := \max\{\beta+1, 4\}$.

Claim. Let $\mathcal{S}^*$ be an optimal solution, i.e., let $\mathcal{S}^*$ be a shortcut assignment that minimizes $\sum_{s,t \in V} \mathcal{V}_{\mathcal{S}^*}(s,t)$. Then it is $\mathcal{S}^* \subseteq C \times \{a\}$.

W.l.o.g it is $q \leq |C|$, which ensures the existence of such a shortcut assignment. Consider the graph G without shortcuts added. There, the search space sizes of all queries from a node in $\widetilde{M}$ to node a sum up to $5M|C|$ as there are $M|C|$ such pairs, each of search space size 5. We now return to the graph $G[\mathcal{S}]$ for an arbitrary shortcut assignment $\mathcal{S}$ with $|\mathcal{S}| = q$. We can bound the value of $\gamma(\mathcal{S}) := \sum_{s \in \widetilde{M}} \mathcal{V}_{\mathcal{S}}(s,a)$ by subtracting the maximal yield for each type of shortcut:

$$\begin{aligned}\gamma(\mathcal{S}) \geq 5M|C| \quad & -1|\mathcal{S} \cap \widetilde{M} \times D_1| \\ & -2|\mathcal{S} \cap \widetilde{M} \times D_2| \quad -M|\mathcal{S} \cap C \times D_2| \\ & -3|\mathcal{S} \cap \widetilde{M} \times \{a\}| \quad -2M|\mathcal{S} \cap C \times \{a\}| \quad -M|\mathcal{S} \cap D_1 \times \{a\}|. \end{aligned} \tag{3.6}$$

Let $\mathcal{S}^* \subseteq C \times \{a\}$ be a shortcut assignment with $|\mathcal{S}^*| = q$. For each shortcut $(c,a) \in \mathcal{S}^*$, we have

$$\mathcal{F}_{(c,d_c^1)}^{\mathrm{can}(\mathcal{V},G[\mathcal{S}^*])}(V_2) = \text{false}\,.$$

Hence $\gamma(\mathcal{S}^*) = \gamma_{opt} := 5M|C| - 2Mq$. Let $\mathcal{S}'$ be a shortcut assignment with $|\mathcal{S}'| = q$ such that $\mathcal{S}' \not\subseteq C \times \{a\}$. With Equation 3.6 we see that $\gamma(\mathcal{S}') \geq \gamma_{opt} + M$ as $M \geq 4$. It holds

$$\sum_{s,t} \mathcal{V}_{\mathcal{S}^*}(s,t) \leq \alpha + \beta + \gamma_{opt} < \alpha + \gamma_{opt} + M \leq \sum_{s,t} \mathcal{V}_{\mathcal{S}'}(s,t)$$

which implies that $\mathcal{S}^*$ is better than $\mathcal{S}'$.

Claim. We can compute in polynomial time an integer B, such that there is a shortcut assignment $\mathcal{S}$ with $|\mathcal{S}| = q$ and $\sum_{s,t\in V} \mathcal{V}_\mathcal{S}(s,t) \leq \min\{B, \alpha+\beta+\gamma_{\text{opt}}\}$ if, and only if, (U,C) contains an exact cover.

Let $\mathcal{S}^*$ be such that $\sum_{s,t\in V} \mathcal{V}_{\mathcal{S}^*}(s,t) \leq \alpha+\beta+\gamma_{\text{opt}}$. Then $\mathcal{S}^* \subseteq C \times \{a\}$ as shown in the previous claim. We write (with some abuse of notation)

$$\sum_{s,t\in V} V_{\mathcal{S}^*}(s,t) = \underbrace{\sum_{s\in V, t\in V\setminus\{a\}} V_{\mathcal{S}^*}(s,t)}_{\alpha} + \Bigg(\underbrace{\sum_{s\in {}^{D_1\cup D_2}_{\cup\{a,h_1,h_2\}}} + \sum_{s\in C}}_{\alpha'} + \sum_{s\in U} + \underbrace{\sum_{s\in\widetilde{M}}}_{=\gamma_{\text{opt}}} \Bigg) V_{\mathcal{S}^*}(s,a)$$

The value of α' is equal for all $\mathcal{S}^* \subseteq C\times\{a\}$ with $|\mathcal{S}^*| = q$ and computable in polynomial time: The corresponding sources in $D_1 \cup D_2 \cup \{a, h_1, h_2\}$ lie behind the shortcuts of $\mathcal{S}^*$ and hence are not affected by the actual choice of $\mathcal{S}^*$. A source $c \in C$ benefits only from $\mathcal{S}^*$ if there is a shortcut (c,a). This holds for exactly q sources, independent of the choice of $\mathcal{S}^*$.

We call a shortcut assignment $\mathcal{S}$ *set-covering* if for each $u \in U$, there is exactly one $c \in C$ such that $u \in c$ and $(c,a) \in \mathcal{S}$. Obviously an exact cover of (C,U) implies a set-covering shortcut assignment of G and vice versa.

It is $\sum_{s\in U} V^*_S(s,a) = 3|U|$ if $\mathcal{S}^*$ is set-covering and greater otherwise: Let u be in U. If S is set-covering, a canonical shortest u-a path in $G[\mathcal{S}^*]$ is of the form s-c-a for a $c \in C$. Therefore, $\mathcal{V}_{(u,v)}(V_2)$ is true, only for one node v which must be in C and for which there is a shortcut (v,a). Because of shortcut (v,a) the flag $\mathcal{V}_{(v,d_v^1)}(V_2)$ is false which implies $V_{\mathcal{S}^*}(u,a) = 3$ which is minimal for every u. On the other hand, if u is not covered by a shortcut, the canonical shortest u-a-path is u-h_1-h_2-a and $V_\mathcal{S}(u,a) = 4$.

Hence with $B := \alpha + \alpha' + 3|U| + \gamma_{\text{opt}}$ follows the claim.

Summary. We have shown that the decision variant of problem ExtShortcutsArcFlags that asks if it is possible to reach a certain search-space size is NP-hard. Hence, also the optimization variant, i.e., problem ExtShortcutsArcFlags is NP-hard. □

3.7 Contraction Hierarchies

The main idea of CONTRACTION HIERARCHIES (CH) [GSSD08] is to further develop the MULTILEVEL OVERLAY GRAPH-approach such that each node has its own level. The technique is based on the notion of a *shortcut*, i.e., an edge (u, v) that is added to G for which $\text{len}(u, v) = \text{dist}(u, v)$. See Section 3.2 for a proper definition. The notation $G[E']$ denotes the graph G with the set E' of shortcuts added.

Given the input graph $G = (V, E, \text{len})$, the degree of freedom is to choose a total order $\prec$ on V. The preprocessing phase then consists of iteratively *contracting* the $\prec$-least node until G is empty. A node v is contracted as follows: For each pair of edges $\{u, v\}$, $\{v, w\}$ such that (u, v, w) is the only shortest u-w-path, a *shortcut* (u, w) is introduced to G. Afterwards v and all of its adjacent edges are removed from G. The output of the preprocessing phase is the graph $H_\prec(G) := G[E']$ where G is the original graph and E' is the set of all shortcuts that were inserted due to node contraction. We call $H_\prec := H_\prec(G)$ the *contraction hierarchy* of G and denote by $|H_\prec|$ the number of edges $|E'|$.

The query is a bidirectional search in $H_\prec$ that only relaxes edges (u, v) with $u \prec v$. Pseudocode of preprocessing phase and query are given as Algorithm 3.7 and Algorithm 3.8. Throughout this section we work on undirected graphs. This is no restriction as the results also hold for directed graphs with edges always being symmetric.

The correctness of the approach basically transfers from the correctness of the Multilevel Overlay Graph-technique: One can see Contraction Hierarchies as a special case of the Multilevel Overlay Graph-technique where each node has its own level. Then, the graph after contraction of a node v is the overlay graph of the next level. For a formal proof of correctness see [GSSD08, Sch08].

Algorithm 3.7: Preprocessing-Phase of CONTRACTION HIERARCHIES

```
input  : graph $G = (V, E, \text{len})$, total order $\prec$ on $V$
output: graph $H_\prec(G) := G'$
1 $G' \leftarrow G$
2 while $V \neq \emptyset$ do
3     $v \leftarrow \prec$-least node in $V$
4     for each pair $(u, v), (v, w) \in E$ do
5         if $(u, v, w)$ is the only shortest $u$-$w$-path in $G$ then
6             $G \leftarrow G[\{(u, w)\}]$                 /* insert shortcut $(u, w)$ */
7             $G' \leftarrow G'[\{(u, w)\}]$
8         $E \leftarrow E \setminus \{(x, y) \in E \mid x = v$ or $y = v\}$     /* remove adjacent edges */
9         $V \leftarrow V \setminus \{v\}$                              /* remove $v$ */
```

The following problem models the preprocessing phase of Contraction Hierarchies analogous to the other techniques in this chapter.

Problem CH Preprocessing. Given a graph $G = (V, E, \text{len})$ and a number $K \in \mathbb{Z}_{\geq 0}$, find an order $\prec$ on V, such that $|H_\prec(G)| \leq K$ and $\sum_{s,t \in V} V_\prec(s, t)$ is minimal?

For this problem, it is not assured that a feasible solution exists: The minimal size $|H_\prec(G)|$ required for the preprocessed data is unknown. We show that already the problem of bounding the size of the preprocessed data is NP-hard. This implies that it is NP-hard to decide if there is a feasible solution for problem CH PREPROCESSING.

Algorithm 3.8: Unidirectional Pruned Search of a CH-Query

```
input : graph $G' = (V, E \cup E', \text{len})$, node $x \in V$, total order $\prec$ on $V$
output: distance label $d()$
1 for $v \in V$ do $d(v) \leftarrow \infty$                    /* Initialization Phase */
2 $d(x) \leftarrow 0$ ; $Q$.INSERT(x,0)
3 while not $Q$.ISEMPTY do                                   /* Main Phase */
4     $v \leftarrow Q$.EXTRACTMIN     /* $v$ is now contained in the search space */
5     for $(v, w) \in E \cup E'$ with $v \prec w$ do
6         if $d(v) + len(v, w) < d(w)$ then
7             $d(w) \leftarrow d(v) + len(v, w)$
8             $Q$.INSERTORUPDATE($w, d(w)$)
```

Problem CH Preprocessing Size. Given a graph $G = (V, E, \text{len})$ and a number $K \in \mathbb{Z}_{\geq 0}$, is there an order $\prec$ on V such that $|H_{\prec}(G)| \leq K$?

Theorem 9. Problem CH PREPROCESSING SIZE is NP-hard.

Proof. We make a reduction from VERTEXCOVER. Given a VERTEX COVER instance $(G = (V, E), K)$, we construct a graph $G' = (V', E', \text{len}')$, which admits a contraction hierarchy H with at most $|E'| + K$ arcs, if and only if, G has a vertex cover of size at most K. From now on let $m = |E|$ and $n = |V|$ and w.l.o.g we assume that each vertex is adjacent to at least one edge. We further may assume that $K < n$ as we otherwise could easily check the solvability of instance (G, k).

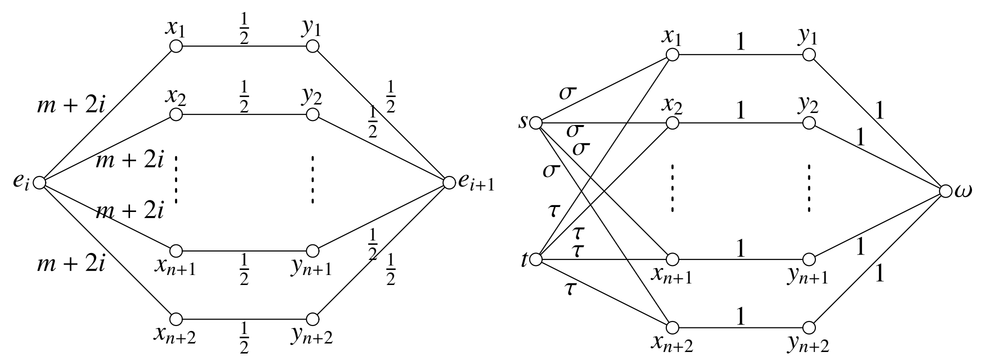
The set $V \cup E$ is a subset of V'. The vertices $E \subset V'$ are henceforth referred to as *edge-vertices*. For each $e = \{u, v\} \in E$ the graph G' contains the edges $\{e, u\} \in E'$ and $\{e, v\} \in E'$. Furthermore V' contains two special vertices $s, t \in V'$, where s is connected to all edge-vertices $e \in E$ and t is connected to all vertices $v \in V$. That is $\{\{s, e\} : e \in E\} \subset E'$ and $\{\{t, v\} | \ v \in V\} \subset E'$.

Now we fix an arbitrary order $e_1, \ldots, e_m$ on E and connect each e_i to e_{i+1} by a *honeycomb gadget* H_i. We later see that H_i enforces the contraction order $e_i \prec e_{i+1}$. The gadget H_i can be seen in Figure 3.12a. Additionally we have a *final gadget* F connecting s and t, which is depicted in Figure 3.12b. Finally we have to fix the edge-lengths in G'. We let $\text{len}(t, v) = \frac{1}{2}m$, $\text{len}(e_i, v) = 2m$ and $\text{len}(s, e_i) = m + i$ for $e_i \in E$ and $v \in V$. The edge-lengths in the gadgets are chosen according to Figure 3.12a and Figure 3.12b. The whole construction is summarized in Figure 3.13. Note that G' can be computed in polynomial time, as K is polynomial in $|V|$.

'*Only if*'. There is a vertex cover $C \subseteq V$ in G of at most K nodes only if G' admits a contraction hierarchy H with at most $K + |E'|$ edges: Let $C \subseteq V$ be a vertex cover with at most K vertices. Consider the following contraction order of V':

1. Contract all $v \in V \setminus C$. This does not insert any shortcuts into the hierarchy: Paths of the form (t, v, e) are no unique shortest paths as there must be a path (t, c, e) of same length with $c \in C$. Paths of the form (e_i, v, e_j) have length $4m$ and are no shortest paths as the path (e_i, s, e_j) has length $2m + i + j < 4m$.

2. Contract all edge-vertices $e \in E$ in the chosen order $e_1, \ldots, e_m$. Note that by contraction of e_i the contraction of the gadget connecting e_i to its successor e_{i+1} is



(a) The honeycomb gadget H_i enforcing contraction order $e_i \prec e_{i+1}$.

(b) The final gadget F connecting s and t. The weights σ and τ are chosen as $\sigma = 5m + \frac{1}{8}$ and $\tau = 5m$.

Figure 3.12: The gadgets used in the reduction from Vertex Cover to CH Preprocessing.

implicitly included. This step inserts at most K shortcuts into the hierarchy. We use the notation from Figure 3.14a.

(a) The path $p = (x_r, e_i, x_s)$ has length $2m + 4i$ and thus is no shortest path, as the path $(x_r, y_r, e_{i+1}, y_s, x_s)$ has length 2.

(b) The path $p = (x_r, e_i, s)$ has length $2m + 3i$, while the path (x_r, y_r, e_{i+1}, s) has length $m + i + 2$, which is, for all $i \geq 1$, less than $2m + 3i$. Therefore p is no shortest path.

(c) The path $p = (x_r, e_i, v)$ has length $3m + 2i$. Again p is no shortest path, as the path $(x_r, y_r, e_{i+1}, v', t, v)$ has length $3m + 1$, which is less than $3m + 2i$ for all $i \geq 1$.

(d) A shortcut (s, v) may be introduced replacing the path $p = (s, e_i, v)$. At this step at most $|C| = K$ vertices are left in V. Hence at most K such shortcuts are introduced.

After contraction of e_i the remaining part of the gadget H_i consists only of the vertex e_{i+1} and simple paths (x_r, y_r, e_{i+1}). Thus it can be contracted without introducing any shortcuts.

To exactly know the structure of G' after this step we additionally proof that there *is* a shortcut (s, v) for each $v \in V$: Let e_i be the first edge-vertex adjacent to v in our fixed order $e_1, \ldots, e_m$, then $p = (v, e_1, s)$ is a unique shortest path of length $3m + 1$ because of the following case distinction.

(a) $p' = (s, e_j, v)$ for some edge-vertex e_j distinct from e_i. Then p' has length $3m + j$, which is greater than $3m + i$ as e_i is the first edge in $e_1, \ldots, e_m$ that is adjacent to v.

(b) $p' = (s, e_j, u, t, v)$ for some edge-vertex $e_j \neq e_i$ and some vertex $u \in V$. Then p' has length $4m + j$, which is greater than $3m + 1$.

(c) $p' = (s, x_r, t, v)$ for some vertex x_r in the final gadget F. Then p' has length at least $10m$, which is greater than $3m + 1$, too.

3. Contract the special vertex s. This does not insert any shortcut in the hierarchy: The remaining graph consists of $\{s, t\} \cup C$, the final gadget F and the set of shortcuts that were inserted $\{\{s, v\} : v \in C\}$, where the edge $\{s, v\}$ has weight $3m + i$, if e_i is

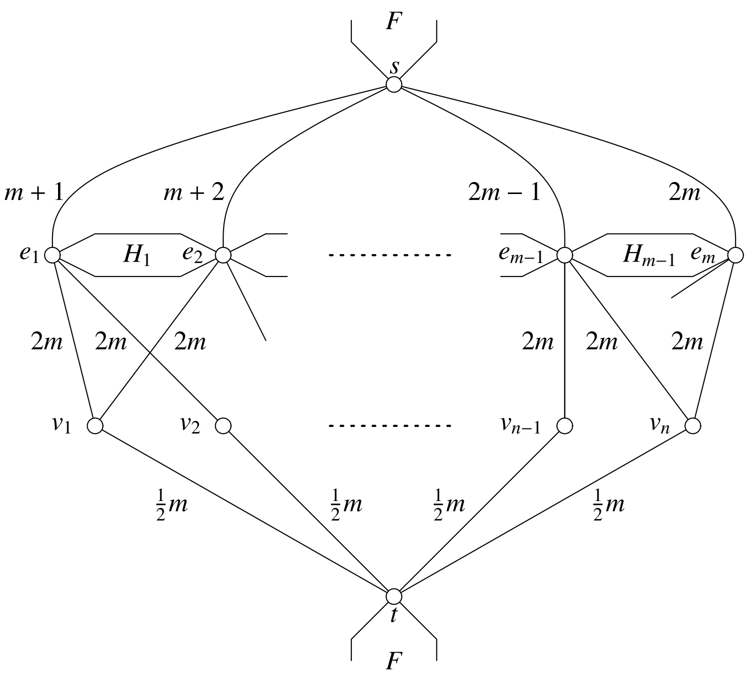


Figure 3.13: Schematic picture of G'. The honeycomb gadgets H_i are depicted by small hexagons between e_i and e_{i+1}. For readability reasons the final gadget F is only shown as half hexagons at s and t.

the first edge-vertex in the order $e_1, \ldots, e_m$ that is adjacent to v. Hence, the graph like the one shown in Figure 3.14b, from which we borrow notation for the following considerations. We have to take the following paths into account:

(a) The path $p = (v, s, v')$ between two vertices $v, v' \in C$ has length greater than $6m$. As the path (v, t, v') has length m the path p is clearly no shortest path.

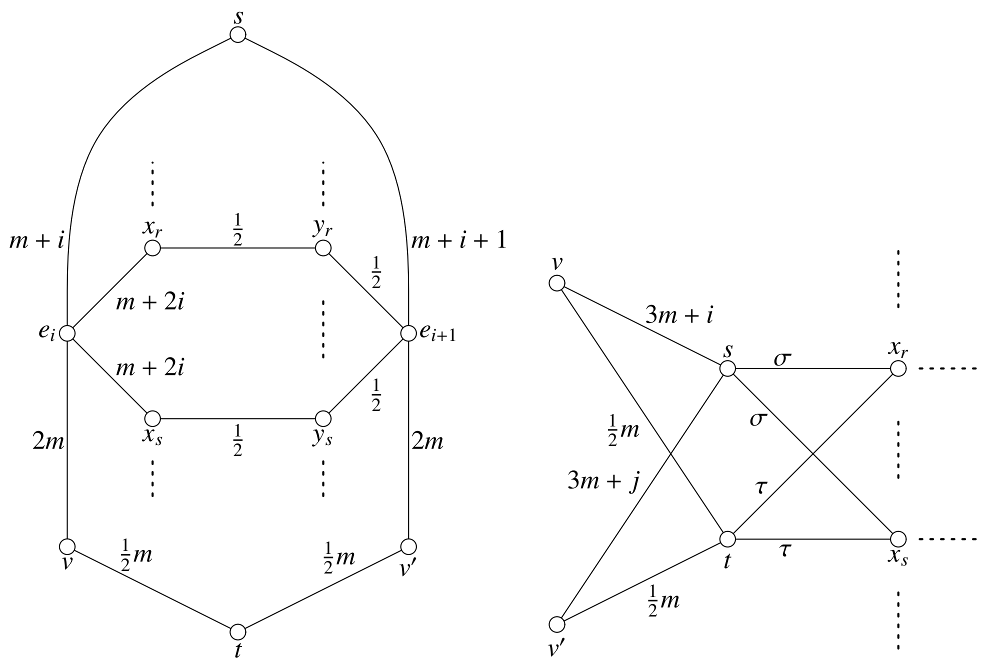
(b) The path $p = (x_r, s, x_s)$ between two vertices x_r and x_s of the final gadget F has length $10m + \frac{1}{4}$, while the path (x_r, t, x_s) has length $10m$. Therefore p is no shortest path.

(c) The path $p = (x_r, s, v)$ has length $8m + i + \frac{1}{8}$. Again, p is no shortest path as the path (x_r, t, v) has length $5m + \frac{1}{2}m$.

4. Contract all $v \in C$. This does not insert any shortcut into the hierarchy as after Step 3 all $v \in C$ have degree one.

5. After Step 4 the remaining graph consists only of the final gadget F without s and its incident edges. For each two distinct vertices x_r, x_s in the final gadget F the path (x_r, t, x_s) has length $10m$. Hence it is no shortest path as $(x_r, y_r, \omega, y_s, x_s)$ has length 4. Thus t can be contracted without introducing any additional shortcuts. After contraction of t the remaining part of F is the vertex ω with paths (x_r, y_r, ω) attached to it. This, too, can be contracted without inserting any new shortcuts into the hierarchy.

'*If*'. On the other hand suppose there is an order $\prec$ on the vertices of G', such that the corresponding contraction hierarchy has at most $|E'| + K$ arcs – or equivalently at most K shortcuts. We will first show some simpler properties that the contraction order $\prec$ must



(a) Contraction of edge-vertex $e_i \in E(G)$.

(b) Contraction of the special vertex s. The edges of weight $m+i$ and $m+j$ originate in the contraction of edge-vertices e_i and e_j. The vertices x_i on the right hand side of the figure are those from the final gadget F. The weights σ and τ were initially chosen as $\sigma = 5m + \frac{1}{8}$ and $\tau = 5m$.

Figure 3.14: Important steps during contraction of G' given a vertex cover $C \subseteq V$.

possess and then construct a vertex cover in G using these properties and the contraction order.

Claim. Each edge-vertex e_i gets contracted before its successor e_{i+1} in the fixed order $e_1, \ldots, e_m$.

Assume the contrary and consider the honeycomb gadget H_i between e_i and e_{i+1}. Without loss of generality let $(x_1, y_1), \ldots, (x_L, y_L)$ be the pairs of vertices (x_r, y_r) in H_i, such that $e_{i+1} \prec x_r$ or $e_{i+1} \prec y_r$. Then there are $n + 2 - L$ pairs $(x_{L+1}, y_{L+1}), \ldots, (x_{n+2}, y_{n+2})$, where $x_r, y_r \prec e_{i+1} \prec e_i$ for $r > L$ which we consider first:

1. $y_r \prec x_r, e_i, e_{i+1}$
   The path $p = (x_r, y_r, e_{i+1})$ is a unique shortest path of length 1:

   (a) The paths $(x_r, e_i, x_s, y_s, e_{i+1})$, where $s \neq r$, have length $2m + 4i + 1$.

   (b) The paths (x_r, e_i, v, e_{i+1}), where v is some vertex $v \in V$ incident to e have length $5m + 2i$.

   (c) The path (x_r, e_i, s, e_{i+1}) has length $3m + 4i + 1$.

2. $x_r \prec y_r, e_i, e_{i+1}$
   The path $p = (e_i, x_r, y_r)$ is a unique shortest path of length $m + 2i + \frac{1}{2}$:

   (a) The paths $(e_i, x_s, y_s, e_{i+1}, y_r)$, where $s \neq r$, have length $m + 2i + \frac{3}{2}$.

   (b) The path (e_i, s, e_{i+1}, y_r) has length $2m + 2i + \frac{3}{2}$.

   (c) The path (e_i, v, e_{i+1}, y_r) has length $4m + \frac{1}{2}$.

Therefore contraction of x_r and y_r before e_i and e_{i+1} results in at least one additional edge being inserted into the hierarchy. This sums up to at least $n+2-L$ additional edges.

Now consider the pairs $(x_1, y_1), \dots, (x_L, y_L)$, where at least one of x_r, y_r gets contracted after e_{i+1}. For $1 \le s \le L$ let z_s be the vertex $z_s \in \{x_s, y_s\}$ that is a neighbour of e_{i+1} when e_{i+1} gets contracted. For distinct z_r, z_s the path $p = (z_s, e_{i+1}, z_r)$ has length at most 2, while paths $(z_r, \dots, e_i, \dots, z_s)$ have length at least $m + 2i$, as they include the edge $\{x_r, e_i\}$ of length $m + 2i$ or the shortcut $\{y_r, e_i\}$ of length $m + 2i + \frac{1}{2}$. Hence p is a unique shortest path and contraction of e_{i+1} before z_s, z_r and e_i inserts an additional shortcut $\{z_r, z_s\}$. As there are $\frac{1}{2}L(L-1)$ such pairs $\{z_r, z_s\}$, contraction of e_{i+1} leads to the insertion of $\frac{1}{2}L(L-1)$ shortcuts.

Altogether, contraction of e_{i+1} before e_i results in at least $n+2-L+\frac{1}{2}L(L-1) \ge n > K$ shortcuts contradicting the assumption of at most K inserted edges.

Claim. The special vertices s and t get contracted before the vertex ω in the final gadget F.

Assume the contrary, i.e., that $\alpha \in \{s, t\}$ gets contracted after ω. Further let $\alpha' \in \{s, t\} \setminus \{\alpha\}$. Partition the pairs (x_r, y_r) of vertices in F, such that $\omega \prec x_r$ or $\omega \prec y_r$ for all $1 \le r \le L$ and such that $x_r, y_r \prec \omega$ for all $L+1 \le r \le n+2$. Now consider the following contraction orders:

1. $y_r \prec x_r, \omega, \alpha$. In this situation (x_r, y_r, ω) is a unique shortest path.

2. $x_r \prec y_r, \omega, \alpha$. In this situation (α, x_r, y_r) is a unique shortest path.

Contraction of x_r and y_r before α and ω inserts in any case at least one shortcut which sums up to at least $n + 2 - L$ additional edges in the hierarchy.

Let z_s for $1 \le s \le L$ be the vertex $z_s \in \{x_s, y_s\}$ that is a neighbour of ω when ω gets contracted. For distinct z_s, z_r the path $p = (z_s, \omega, z_r)$ has length at most 4. As any path (z_s, α, z_r) or (z_s, α', z_r) has length at least $10m$, p is a unique shortest path. Contraction of ω before z_s, z_r therefore inserts an additional shortcut $\{z_s, z_r\}$. As there are $\frac{1}{2}L(L-1)$ such pairs $\{z_s, z_r\}$, contraction of ω inserts at least $\frac{1}{2}L(L-1)$ shortcuts.

Altogether, contraction of ω before α results in at least $n+2-L+\frac{1}{2}L(L-1) \ge n > K$ additional shortcuts being inserted. This is a contradiction and thus $\alpha \prec \omega$.

Claim. The special vertex t gets contracted after all $v \in V$.

Assume the contrary and let $v_0 \in V$ be a vertex with $t \prec v_0$. By the last claim we may assume that ω is still present when t gets contracted. Consider the final gadget F and partition the pairs (x_r, y_r) of vertices in F, such that $t \prec x_r$ or $t \prec y_r$ for all $1 \le r \le L$ and such that $x_r, y_r \prec t$ for all $L+1 \le r \le n+2$. Now consider the following contraction orders:

1. $y_r \prec x_r, t, \omega$. In this situation (x_r, y_r, ω) is a unique shortest path of length 2.

2. $x_r \prec y_r, t, \omega$. In this situation (t, x_r, y_r) is a unique shortest path of length $5m + 1$.

Contraction of x_r and y_r before t hence inserts at least one additional edge into the hierarchy which sums up to at least $n + 2 - L$ additional shortcuts. For $1 \le r \le L$ now let z_r be the vertex $z_r \in \{x_r, y_r\}$ that is adjacent to t, when t gets contracted. We have $\operatorname{dist}(v_0, s) \ge 3m + 1$, $\operatorname{dist}(s, x_r) \ge 5m + \frac{1}{8}$ and $\operatorname{dist}(s, y_r) \ge 5m + \frac{9}{8}$.

1. Let $z_r = x_r$ then $p = (v_0, t, x_r)$ is a unique shortest path of length $\frac{11}{2}m$.

2. Let $z_r = y_r$ then $p = (v_0, t, x_r, y_r)$ is a unique shortest path of length $\frac{11}{2}m + 1$.

Contraction of t therefore results in the insertion of an additional shortcut $\{v_0, z_r\}$. As there are L such neighbours z_r of t, contraction of t inserts at least L additional edges. Altogether contraction of v after t results in $n+2-L+L \geq n > K$ additional shortcuts, which is a contradiction.

Claim. All edge-vertices $e_i \in E$ get contracted before s.

Assume the contrary, i.e., that there is some edge-vertex $e_i \in E$ that gets contracted after s. Consider the final gadget F and partition the pairs (x_r, y_r) of vertices in F, such that for all $1 \leq r \leq L$ it is $s \prec x_r$ or $s \prec y_r$ and such that for all $L+1 \leq r \leq n+2$ it is $x_r, y_r \prec s$. By the last claims we know that $x_r \prec s$ and $y_r \prec s$ imply $x_r \prec \omega$ and $y_r \prec \omega$ respectively. Now consider the following contraction orders.

1. $y_r \prec x_r, s, \omega$. In this situation (x_r, y_r, ω) is a unique shortest path of length 2.

2. $x_r \prec y_r, s, \omega$. In this situation (s, x_r, y_r) is a unique shortest path of length $5m+\frac{1}{8}+1$.

Contraction of x_r and y_r before s hence inserts at least one additional edge into the hierarchy which sums up to at least $n+2-L$ additional edges into the hierarchy.

For $1 \leq r \leq L$ let z_r be the vertex $z_r \in \{x_r, y_r\}$ that is adjacent to s, when s gets contracted. The path $p = (e_i, s, z_r)$ has length $6m + i + \frac{1}{8}$ for $z_r = x_r$ and length $6m + i + \frac{1}{8} + 1$ for $z_r = y_r$. The path $p' = (e_i, u, t, x_r)$ in G', where u is some vertex $u \in V$, has length $7m + \frac{1}{2}m$ and $p' = (e_i, u, t, x_r, y_r)$ in G' has length $7m + \frac{1}{2}m + 1$. For p is a unique shortest path in G', p is a unique shortest path, when s gets contracted, too. Contraction of s therefore inserts a shortcut $\{e_i, z_r\}$ into the hierarchy. As there are L such vertices z_r contraction of s results in the insertion of at least L such shortcuts.

Altogether contraction of e_i after s resulted in $n+2-L+L \geq n > K$ additional shortcuts, which is a contradiction.

Subsumption. The following observations subsume the above claims about possible pair-wise contraction orders.

1. E gets contracted in order $e_1, \ldots, e_m$.

2. $V \prec t$ and $E \prec s$, that is whenever we encounter vertices $v \in V$ or edge-vertices $e \in E$, we may assume that the vertex t or the vertex s respectively are not contracted yet.

In the final step of this proof we will construct a vertex cover for the original graph G and prove that it contains at most K vertices.

For each vertex $v \in V$, let $e_{\min}(v)$ be the first edge-vertex in order $e_1, \ldots, e_m$ that is incident to v, that is $e_{\min}(v) = e_M$, where $M = \min\{i : e_i \text{ is incident to } v\}$. We partition E into two sets. The set E_1 contains those edges that are incident to some vertex v that gets contracted after $e_{\min}(v)$.

$$E_1 = \{e = \{u, v\} \in E |\ e_{\min}(u) \prec u \text{ or } e_{\min}(v) \prec v\}$$

Secondly we let E_2 be the set of edges that are incident to two vertices u and v that get both contracted before $e_{\min}(v)$.

$$E_2 = \{e = \{u, v\} \in E |\ u \prec e_{\min}(u) \text{ and } v \prec e_{\min}(v)\}$$

Obviously it is $E = E_1 \dot{\cup} E_2$. Now we define for each edge $e \in E$ the *cover vertex* $v(e)$ of e as follows:

$$v(e) = \begin{cases} \text{arbitrary } u \in V \text{ incident to } e \text{ in } G \text{ such that } e_{\min}(u) \prec u & , e \in E_1 \\ \prec\text{-maximal } u \in V \text{ incident to } e \text{ in } G & , e \in E_2 \end{cases}$$

Claim. $C = \{v(e) : e \in E\}$ is a vertex cover in G.

Let $e = \{u, v\} \in E$. Then $v(e) = u$ or $v(e) = v$ by definition of the cover vertex $v(e)$.

Claim. $C = \{v(e) : e \in E\}$ has size at most K.

As there are at most K shortcuts in the contraction hierarchy, it suffices to show that there is an injective mapping $M \colon C \to S$, where S is the set of shortcuts inserted during the contraction of G' using contraction order $\prec$. We will construct M by assigning to each vertex $v \in v(E_1)$ the shortcut $\{s, v\}$ and to each $v \in v(E_2)$ a shortcut of the form $\{t, e\}$.

Observe that $v(E_1)$ and $v(E_2)$ are disjoint, as $u \in v(E_1) \cap v(E_2)$ would imply $e_{\min}(u) \prec u \prec e_{\min}(u)$. Since the shortcuts assigned to $v \in v(E_1)$ and $v \in v(E_2)$ are of different kind, it is clear that $M \colon C \to S$ is well-defined and injective on $C = v(E_1) \cup v(E_2)$, if it is well-defined and injective on $v(E_1)$ and $v(E_2)$.

First consider a vertex $v = v(e) \in C$ with $e \in E_1$. Then, by definition of E_1, $v \succ e_i = e_{\min}(v)$. By the subsumption we know that s is still present in the graph when e_i gets contracted. Now consider possible paths between s and v at this step.

1. The path $p = (s, e_i, v)$ has length $3m + i$. For any other e_j the path (s, e_j, v) has length $3m + j$ and since $e_i = e_{\min}(v)$ the path p is a unique shortest path among the paths (s, e_j, v).

2. For some vertex $u \neq v$ and some edge-vertex $e_j \neq e_i$ the path (s, e_j, u, t, v) has length $4m + j$.

3. The path (s, x_r, t, v), where x_r is a vertex of the final gadget F, has length $(21/2)m + \frac{1}{8}$.

Hence, (s, e_i, v) is a unique shortest path when e_i gets contracted and thus contraction of e_i inserts a shortcut $\{s, v\}$. We set $M(v) := \{s, v\}$.

Next we account for the vertices in $v(E_2)$. Let $v \in v(E_2)$. Choose arbitrary $e = \{u, v\}$ in E_2 such that $v = v(e)$. By definition of E_2, we have $u \prec e_{\min}(u)$ and $v \prec e_{\min}(v)$. In particular $u \prec v \prec e$. By the subsumption, the vertex t is not contracted, when u or v get contracted. Consider the paths $p_u = (t, u, e)$ and $p_v = (t, v, e)$, each of length $\frac{5}{2}m$. Apart from p_u and p_v the only relevant path between t and e is $p' = (t, x_r, s, e)$, where x_r is some vertex of the final gadget F. The length of p' is greater than $10m$ and thus p_u and p_v are shortest paths.

When v gets contracted, u and the path p_u are already contracted and p_v is a unique shortest path. Contraction of v hence inserts a shortcut $\{t, e\}$ into the hierarchy. We set $M(v) := \{t, e\}$. Observe that M is injective on $v(E_2)$, as $M(x) = M(y)$ implies $x = y = v(e')$ for some e'. This finishes the proof. □

3.8 Lower Bounds for Search-Space Guarantees

We now consider the question of how good a speedup-technique actually can get: Let preprocessing time and space be unrestricted, what guarantee can a speedup-technique give for the average search space size? We focus on guarantees that depend on the size of the input graph.

The situation is clear for techniques that do not insert shortcuts into the graph. When considering a path as the input graph, it is easy to see that no guarantee better than the trivial $O(|V|)$-guarantee is possible. However, Arc-Flags and ALT are able to encode all-pairs shortest-paths in their preprocessed data. Hence, it is possible to show that both techniques can guarantee to settle at most all nodes that lie on a shortest s-t-path.

The situation is not so clear for the Multilevel Overlay Graph-technique, Contraction Hierarchies, Arc-Flags with external shortcuts and Reach Based Pruning with external shortcuts. The main contribution of this section focuses on Contraction Hierarchies. When working with general graphs, the situation is clear: Considering the complete graph with $|V|$ nodes we observe that there is no guarantee better than the trivial $O(|V|)$ guarantee. Hence, we focus on graphs with bounded degree and on trees and give a lower bound of $\Omega(\operatorname{ld}|V|)$ for any guarantee on the average search space size. This bound is tight when restricting the input to paths.

Throughout the remainder of this section, we denote by $P_n = (V_n, E_n, \mathrm{len})$ a path with n nodes and uniform edge lengths, i.e.,

$$V_n = \{1, \dots, n\} \qquad E_n = \{\{i, i+1\} \mid 1 \leq i < n\} \qquad \mathrm{len} \equiv 1 \ .$$

Techniques without Shortcuts. Obviously, techniques that do not insert shortcuts into the graph contain at least as much nodes in the s-t-search space as a shortest s-t-path requires. We consider the average search space of a path P_n. The sum of sizes of all shortest paths in P_n is $2(\sum_{i=1}^{n}\sum_{j=1}^{i} j) - n$ as we consider only paths from node i to nodes $t \leq i$ and by symmetry reasons multiply this by 2. We subtract n as we count the n pairs with $i = t$ twice. We simply sum over the sizes of all shortest s-t-paths to obtain the following bound.

$$\begin{aligned}
\frac{1}{n^2}\sum_{i,t=1}^{n} \mathcal{V}(i,t,P_n) &\geq \frac{1}{n^2}\left(2\sum_{i=1}^{n}\sum_{j=1}^{i} j - n\right) \\
&= \frac{1}{n^2}\left(\sum_{i=1}^{n}(i^2+i)\right) - \frac{1}{n} \\
&= \frac{1}{n^2}\left(\frac{n(n+1)(2n+1)}{6} + \frac{n(n+1)}{2}\right) - \frac{1}{n} \\
&= \frac{n+1}{n}\left(\frac{2n+1}{6} + \frac{1}{2}\right) - \frac{1}{n} = \Omega(n) \ .
\end{aligned}$$

Techniques that apply shortcuts can break this barrier. For a brief glance at techniques without shortcuts, we include the number $|P^{\bowtie}(s,t)|$ of vertices that lie on a shortest s-t-path into our considerations. We do this for ALT and ARC-FLAGS.

Arc-Flags. An optimal unrestricted solution for Arc-Flags is obvious: Each node is its own cell. When working with vector $\mathcal{F}^{\mathrm{all}(\mathcal{V},G)}$ this yields the guarantee $\mathcal{V}(s,t,G) \subseteq P^{\bowtie}(s,t)$ for any pair $s,t \in V$ with $\operatorname{dist}(s,t) < \infty$. When working with vector $\mathcal{F}^{\mathrm{can}(\mathcal{V},G)}$ the value of $|\mathcal{V}(s,t,G)|$ improves to the minimal number of nodes on a shortest s-t-path.

ALT. The set of landmarks $L = V$ not necessarily is optimal but gives the guarantee $\mathcal{V}(s,t,G) \subseteq P^{\bowtie}(s,t)$ for any pair $s,t \in V$ with $\operatorname{dist}(s,t) < \infty$:

If $\operatorname{dist}(v,t) = \infty$ it is $\Pi_t^L(v) \geq \Pi_t^{v-}(v) = \infty$. With Lemma 6 follows that v is not in the s-t-search space. Further, for $v \notin P^{\bowtie}(s,t)$ with $\operatorname{dist}(v,t) < \infty$ we have that $\operatorname{dist}(s,v) + \operatorname{dist}(v,t) > \operatorname{dist}(s,t)$. This yields $\operatorname{dist}(s,v) + \Pi_t^L(v) \geq \operatorname{dist}(s,v) + \Pi_t^{v-}(v) = \operatorname{dist}(s,v) + \operatorname{dist}(v,t) - \operatorname{dist}(v,v) > \operatorname{dist}(s,t)$. With Lemma 6 follows that v is not in the s-t-search space.

Contraction Hierarchies. For arbitrary graphs, no bound outside $\Omega(|V|)$ is possible as the complete graph with $|V|$ nodes and uniform edge lengths has average search space size $(|V|+1)$. This follows from

$$\begin{aligned}
\frac{1}{|V|^2} \sum_{s,t \in V} \mathcal{V}(s,t) &= \frac{1}{|V|^2} |V| \sum_{z \in V} \Big(\mathcal{V}^+(z) + \mathcal{V}^-(z) \Big) \\
&= \frac{2}{|V|} \sum_{z \in V} \Big(|\{v \in V : z \prec v \vee z = v\}| \Big) = |V| + 1 \ .
\end{aligned}$$

Hence, in the following we consider sparse graphs. We still work on undirected graphs and, for simplicity, consider only one direction of the actual query. We show that the optimal average search space size achieved on a path is logarithmic in $|V|$. This gives a lower bound of $\Omega(\operatorname{ld}|V|)$ for any guarantee on the average search space size on a large class of sparse graphs, especially for graphs with bounded degree or trees. Throughout the remainder of this section, we consider the input graph to be a path $P_n = (V_n, E_n, \mathrm{len})$.

We approach the proof as follows: We first conjecture that the *sorting number* $B(n+1)$ gives a lower bound for the sum of the search space sizes on a path with n nodes. We then give two auxiliary results for the sorting numbers and afterwards proof the conjecture. Next, we present an algorithm that shows that the bound is tight on paths. Finally, we evaluate the asymptotic behavior of the sorting numbers which yields our claim.

Given a graph $G = (V, E, \mathrm{len})$, an order $\prec$ on V and the contraction hierarchy $(V, E^*, \mathrm{len}) := H_\prec(G)$, the *directed contraction hierarchy* $\overrightarrow{H}(G,\prec)$ is defined as

$$\overrightarrow{H}(G,\prec) := (V, \{(u,v) \mid \{u,v\} \in E^*, u \prec v\}).$$

Remember that $\operatorname{dist}_G(u,v)$ denotes the distance from vertex u to vertex v in graph G. We define

$$\mathcal{V}_{G,\prec}(u) := \{v \in V_n \mid \operatorname{dist}_{\overrightarrow{H}(G,\prec)}(u,v) < \infty\}$$

and consequently, $|\mathcal{V}_{G,\prec}(u)|$ is the number of nodes visited during a query starting at vertex u. Further,

$$\mathcal{V}(G,\prec) := \sum_{u \in V} |\mathcal{V}_{G,\prec}(u)|$$

is n times the average search space size (of one direction) for the contraction hierarchy $H_\prec(G)$. For all definitions, we will leave out the order $\prec$ whenever the choice of $\prec$ is clear. We further use the convention $\sum_{i=1}^{0} f(i) := 0$ for any function f.

Lemma 11. For all $n \in \mathbb{Z}_{>0}$ and all orders $\prec$ on V_n, it is

$$\mathcal{V}(P_n,\prec) \geq B(n+1)$$

where $B(k) = \sum_{i=1}^{k} \lceil \operatorname{ld} i \rceil$ and $P_n = (V_n, E_n)$ with $V_n = \{1, \ldots, n\}$ and $E_n = \{\{i, i+1\} \mid 1 \leq i < n\}$.

In order to proof this result we first give two facts on the sequence $B(k)$. The *sorting numbers* $B(k)$ are sequence A001855 in [Slo08]. The following recursive formula is a key to the proof of Lemma 11.

Lemma 12. Let $B(k) = \sum_{i=1}^{k} \lceil \mathrm{ld}\, i \rceil$ for $k \in \mathbb{Z}_{\geq 0}$. Then

$$B(n) = B\left(\left\lceil \frac{n}{2} \right\rceil\right) + B\left(\left\lfloor \frac{n}{2} \right\rfloor\right) + n - 1$$

Proof. We do induction on n. The case $n = 1$ is easy to check. For $n = 2$ we have

$$B(2) = 0 + 1 = B\left(\left\lfloor \frac{2}{2} \right\rfloor\right) + B\left(\left\lceil \frac{2}{2} \right\rceil\right) + 2 - 1$$

For $n > 2$, we get by induction hypothesis

$$B(n) = B(n-1) + \lceil \mathrm{ld}\, n \rceil = B\left(\left\lfloor \frac{n-1}{2} \right\rfloor\right) + B\left(\left\lceil \frac{n-1}{2} \right\rceil\right) + \lceil \mathrm{ld}\, n \rceil + n - 2$$

If n is even, then $\lfloor \frac{n-1}{2} \rfloor = \frac{n}{2} - 1$ and $\lceil \frac{n-1}{2} \rceil = \frac{n}{2}$. Furthermore $\lceil \mathrm{ld}\, n \rceil = \lceil \mathrm{ld}\, \frac{n}{2} \rceil + 1$ and thus we get

$$B(n) = B\left(\frac{n}{2} - 1\right) + B\left(\frac{n}{2}\right) + \lceil \mathrm{ld}\, n \rceil + n - 2 = B\left(\frac{n}{2}\right) + B\left(\frac{n}{2}\right) + n - 1$$

If n is odd, then $\lceil \frac{n-1}{2} \rceil = \frac{n-1}{2} = \lfloor \frac{n-1}{2} \rfloor$. Additionally $\lceil \mathrm{ld}\, n \rceil = \lceil \mathrm{ld}\, \frac{n+1}{2} \rceil + 1$ and we get

$$\begin{aligned} B(n) &= B\left(\frac{n-1}{2}\right) + B\left(\frac{n-1}{2}\right) + \left\lceil \mathrm{ld}\, \frac{n+1}{2} \right\rceil + n - 1 \\ &= B\left(\frac{n-1}{2}\right) + B\left(\frac{n+1}{2}\right) + n - 1 \end{aligned}$$

This finishes the proof. □

Further, from the monotonicity of ld we have the following inequality.

Lemma 13. Let $B(k) = \sum_{i=1}^{k} \lceil \mathrm{ld}\, i \rceil$ for $k \in \mathbb{Z}_{\geq 0}$. Further, let $n_1, n_2, n \in \mathbb{Z}_{\geq 0}$ with $n_1 + n_2 = n$. Then

$$B(n_1) + B(n_2) \geq B\left(\left\lceil \frac{n}{2} \right\rceil\right) + B\left(\left\lfloor \frac{n}{2} \right\rfloor\right)$$

Proof. Without loss of generality let $n_1 \geq n_2$. As $n_1 + n_2 = n$ this implies in particular $n_1 \geq \lceil \frac{n}{2} \rceil$ and $\lfloor \frac{n}{2} \rfloor \geq n_2$. The special case $n_2 = 0$ is easy to check. Now this lemma follows directly from the monotonicity of ld and the definition of sorting numbers:

$$\begin{aligned} B(n_1) + B(n_2) &= \sum_{i=1}^{n_1} \lceil \mathrm{ld}\, i \rceil + \sum_{i=1}^{n_2} \lceil \mathrm{ld}\, i \rceil = \sum_{i=1}^{\lceil \frac{n}{2} \rceil} \lceil \mathrm{ld}\, i \rceil + \sum_{i=\lceil \frac{n}{2} \rceil + 1}^{n_1} \lceil \mathrm{ld}\, i \rceil + \sum_{i=1}^{n_2} \lceil \mathrm{ld}\, i \rceil \\ &\geq B\left(\left\lceil \frac{n}{2} \right\rceil\right) + \sum_{i=n_2+1}^{n_2+n_1-\lceil \frac{n}{2} \rceil} \lceil \mathrm{ld}\, i \rceil + \sum_{i=1}^{n_2} \lceil \mathrm{ld}\, i \rceil = B\left(\left\lceil \frac{n}{2} \right\rceil\right) + B\left(\left\lfloor \frac{n}{2} \right\rfloor\right) \end{aligned}$$

This was to show. □

Equipped with these two lemmata on the sorting numbers we now approach the proof of Lemma 11:

Proof (of Lemma 11). We do induction on n. If $n = 1$, there is nothing to show, as $B(2) = 1$. Now let $n > 1$. Furthermore let $\prec$ be an order on V_n and v be the $\prec$-largest vertex in V_n. Removal of v splits P_n into graphs P_1, P_2 and $H = H_\prec(p_n)$ into graphs H_1, H_2 of n_1 and n_2 vertices, where $n_1 + n_2 = n - 1$. It is $H_1 = H_\prec(P_1)$ and $H_2 = H_\prec(P_2)$. Furthermore $\mathcal{V}_{P_i}(v) = \mathcal{V}_{P_n}(v) - 1$ for all vertices $v \in V(P_i)$ and thus

$$\mathcal{V}(P_n) = \mathcal{V}(P_1) + n_1 + \mathcal{V}(P_2) + n_2 + 1$$

By induction hypothesis we have $\mathcal{V}(P_i) \geq B(n_i + 1)$. Hence we may apply Lemmata 12 and 13 to obtain

$$\begin{aligned}\mathcal{V}(P_n) &\geq B(n_1 + 1) + B(n_2 + 1) + n \\ &\geq B\left(\left\lceil \frac{n+1}{2} \right\rceil\right) + B\left(\left\lfloor \frac{n+1}{2} \right\rfloor\right) + n = B(n+1)\end{aligned}$$

which was to show. □

The lower bound of $B(n+1)$ is tight as Algorithm 3.9 computes an according order $\prec$ on V_n.

Lemma 14. Given input P_n, Algorithm 3.9 computes an order $\prec$ on V_n such that such that $\mathcal{V}(P_n, \prec) = B(n + 1)$.

The proof is by induction analogous to the proof of Lemma 11.

Proof. We do induction on n. The case $n = 1$ holds as $\mathcal{V}(P_n, \prec) = 1 = B(2)$. Now let $n > 1$. Then

$$\begin{aligned}\mathcal{V}(P_n) &= \mathcal{V}(P_{\lfloor (n-1)/2 \rfloor}) + \left\lfloor \frac{n-1}{2} \right\rfloor + \mathcal{V}(P_{\lceil (n-1)/2 \rceil}) + \left\lceil \frac{n-1}{2} \right\rceil + 1 \\ &= B(\left\lfloor \frac{n-1}{2} \right\rfloor + 1) + B(\left\lceil \frac{n-1}{2} \right\rceil + 1) + n \\ &= B(n+1)\end{aligned}$$

This was to show. □

Algorithm 3.9: OptimalPathOrder

```
Input  : Path $P_n = (V_n, E_n)$ of $n$ vertices
Output: Order $\prec$ on $V_n$, such that $\mathcal{V}(H_\prec) = B(n+1)$
if $n = 0$ then
    return $\langle\ \rangle$;
else
    Pick vertex $v \in V$ separating $P_n$ into paths $Q$ and $R$ of length $\lfloor \frac{n-1}{2} \rfloor$ and $\lceil \frac{n-1}{2} \rceil$;
    return OptimalPathOrder($Q$) $\circ$ OptimalPathOrder($R$) $\circ$ $\langle v \rangle$;
```

Corollary. It is $\mathcal{V}(P_n, \prec_n) = \Omega(n \operatorname{ld} n)$ for all orders $\prec_n$ on V_n and there are orders $\prec_n$ on V_n, such that $\mathcal{V}(P_n, \prec_n) = \Theta(n \operatorname{ld} n)$.

Proof. We first show by induction on n that $B(n) = n\lceil \operatorname{ld} n\rceil - 2^{\lceil \operatorname{ld} n\rceil} + 1$. If $n = 2$ we have $B(2) = 1 = 2 - 2 + 1$. If $n > 2$, then

$$\begin{aligned} B(n) &= B(n-1) + \lceil \operatorname{ld} n\rceil \\ &= (n-1)\lceil \operatorname{ld}(n-1)\rceil - 2^{\lceil \operatorname{ld}(n-1)\rceil} + 1 + \lceil \operatorname{ld} n\rceil \end{aligned}$$

If $\lceil \operatorname{ld} n - 1\rceil = \lceil \operatorname{ld} n\rceil$, the claimed equality follows immediately. On the other hand, $\lceil \operatorname{ld} n - 1\rceil < \lceil \operatorname{ld} n\rceil$, if and only if $n = 2^k + 1$ for some $k \in \mathbb{Z}_{\geq 0}$. In that case we have

$$\begin{aligned} B(n) &= (n-1)\lceil \operatorname{ld}(n-1)\rceil - 2^{\lceil \operatorname{ld}(n-1)\rceil} + 1 + \lceil \operatorname{ld} n\rceil \\ &= (n-1)\cdot k - 2^k + 1 + k + 1 \\ &= nk - n + 3 \\ &= n(k+1) - 2(n-1) + 1 \\ &= n(k+1) - 2^{k+1} + 1 \\ &= n\lceil \operatorname{ld} n\rceil - 2^{\lceil \operatorname{ld} n\rceil} + 1 \end{aligned}$$

which shows $B(n) = n\lceil \operatorname{ld} n\rceil - 2^{\lceil \operatorname{ld} n\rceil} + 1$. The corollary is an immediate consequence of

$$\begin{aligned} n\lceil \operatorname{ld} n\rceil - 2^{\lceil \operatorname{ld} n\rceil} + 1 &\leq n(\operatorname{ld} n + 1) + 1 = O(n \operatorname{ld} n) \\ n\lceil \operatorname{ld} n\rceil - 2^{\lceil \operatorname{ld} n\rceil} + 1 &\geq \quad n \operatorname{ld}(n) - 2n = \Omega(n \operatorname{ld} n) \end{aligned}$$

which means $B(n) = \Theta(n \operatorname{ld} n)$. □

3.9 Conclusion

Speedup-techniques have been widely studied experimentally for the last years, especially for road networks. There is large interest in a theoretical foundation of the techniques developed and some first work on the topic has been published [AFGW10, BDDW09].

Complexity Status of the Preprocessing Phase. In this chapter we focused on the preprocessing phases of the recent techniques. These usually incorporate a degree of freedom that, in practice, is filled in a heuristical manner. The quality of a preprocessing strategy can be measured by the effort of the according query. We used the average number of settled nodes in an s-t-query of random nodes s and t as such a measure.

Until now, the complexity status of filling the according degree of freedom was unknown. We settled this question by showing that all variants considered are NP-hard to optimize. This prepared the ground for further work on the given problems.

There are numerous open questions for the topic. A reasonable next step is the development of approximation- or fixed parameter tractable algorithms for the preprocessing phase. Efficient algorithms for special graph classes would help to show the bounds of intractability and to further understand the approaches. These might also be used to yield runtime guarantees for the queries of the given techniques. When working on speedup-techniques that have shown to be hard on directed acyclic graphs it would be particularly interesting to know the complexity status on trees. Finally, we have the conjecture –but did not prove– that problem ArcFlags is NP-hard already for two cells.

A Critical View on the Applied Models. Theoretically modeling a sophisticated algorithm-engineering approach goes with taking many decisions 'in a reasonable way'. The decision to use the average search space size as objective function is motivated by the common practice to compare experimental approaches by query runtime and search space size. Should we measure the search space in terms of settled nodes (which emphasizes the queue operations) or is it better to count the number of relaxed edges (which emphasizes the edge relaxations)? We have positively tested, for some of the proofs in this chapter, if an adaption to 'edge relaxations' is possible. Accordingly, we expect to obtain the same results for that model with slightly modified proofs.

We also briefly considered to optimize the worst-case search space. It seems much harder to adapt our proofs for that model. However, we expect that these problems are also NP-hard and positively tested for problem ArcFlags.

Further, many of the considered approaches consist of a large number of heuristic improvements. Some of these are not even documented in the corresponding paper but hidden inside the program code. We considered some of these improvements separately, such as External Shortcuts for Reach-Based Pruning or Search-Space Minimal Reach. Many other improvements were just neglected. Sometimes reasons for ignoring part of a technique are given in the corresponding paper. An example is the relaxation of the stopping criterion of Contraction Hierarchies: This was already mentioned to be reasonable in [GSSD08]. For others we had to make our own experiments for which we mainly used the code basis of [BDW07]. An example is the relaxation of the stopping criterion of Reach-Based Pruning. Furthermore, we often talked to the programmers of the corresponding approach to gain more knowledge on the impact of a specific improvement.

Summarizing, we claim the considered models to be reasonable. However, we think that the model of the ALT-algorithm can be improved, a more detailed discussion is given in the corresponding section.

Lower Bounds for the Search Space Size of Contraction Hierarchies. We also considered the question of how good a speedup-technique can actually get: Let preprocessing time and space be unrestricted, what guarantee can a speedup-technique give for the size of the average search space? We focused on guarantees that solely depend on the input size, which is the most fundamental measure for such considerations.

The situation is clear for techniques that do not use shortcuts: Guarantees for such approaches lie in $\Omega(|V|)$ which is not better than Dijkstra's algorithm. However, ALT and Arc-Flags can encode all-pairs shortest-paths in a way such that the search-space of an s-t-query consists only of nodes that lie on a shortest s-t-path. Even stronger, Arc-Flags can guarantee that the search-space of an s-t-query consists only of a node-minimal shortest path.

The situation is less obvious for the Multilevel Overlay Graph-technique, Reach-Based Pruning with Shortcuts, Arc-Flags with Shortcuts and Contraction Hierarchies as these could break the $\Omega(|V|)$ barrier. For Contraction Hierarchies we presented a concrete answer. We have shown that any guarantee for the average search space of an s-t-query lies in $\Omega(|V|)$ for arbitrary graphs and in $\Omega(\log|V|)$ for graphs with bounded degree and for trees. Is the second bound tight, i.e., are there algorithms that guarantee the average search space to be in $O(\log|V|)$? We further conjecture that the same or similar bounds hold for the other techniques that apply shortcuts and leave this as an open question.

Analyzing guarantees that solely depend on the input size is a reasonable and insightful point of view. However, our work indicates that a deeper understanding can be gained from applying another measure. In Chapter 4.6 we define the *shortest paths diameter* $\mathrm{spDiam}(G)$ to be the maximum hop-distance between any two nodes in the graph G. This could be a good parameter for future work on such guarantees. Some considerations of this chapter can already be translated to terms of $\mathrm{spDiam}(G)$: For techniques without shortcuts, $\mathrm{spDiam}(G)$ is a lower bound for any guarantee on the s-t-search space of a query. In the unrestricted case, Arc-Flags can guarantee the search space size to be bounded by $\mathrm{spDiam}(G)$. ALT can guarantee the same for graphs in which shortest paths are unique. Contraction Hierarchies can break the $\mathrm{spDiam}(G)$ barrier. However, on arbitrary graphs, no guarantee for the search space size of an s-t-query is possible that depends solely on $\mathrm{spDiam}(G)$. For graphs with bounded degree or trees, any guarantee for the *average* search space size lies in $\Omega(\log \mathrm{spDiam}(G))$.

Models for Road-Networks. Finally, we want to point out an interesting, related branch of research that is not topic of this thesis. Besides railway/timetable-networks, road networks are the main application for route-planning. Accordingly, some of the presented techniques heuristically aim to capture properties of these networks. This can also be done theoretically, related approaches are as follows.

The work [AFGW10] tries to capture one such property by introducing the notion of *highway dimension*. For graphs with low highway dimension, runtime guarantees for some speedup-techniques are given. An obvious and reasonable –yet still open– task is to experimentally evaluate if road networks really exhibit low highway dimension.

Another way to allow for a theoretical analysis that is custom-tailored for road networks is the use of graph generators. We are aware of two generators [AFGW10, BKMW10] that create artifical road networks. Both are experimentally evaluated in [BKMW10]. A reasonable next step into this direction would be to derive helpful properties of graphs originating from these generators.

Some related ideas are given in [EG08] by discussing the question if road networks are almost planar as often claimed. In [EG08] this claim is refused and a model for road networks is given. This model can be used for deriving further properties but has some open degrees of freedom such that it cannot directly be used as a generator.

Chapter 4

The Shortcut Problem

We study a graph-augmentation problem arising from a technique applied in recent approaches for route planning. Many such methods enhance the graph by inserting *shortcuts*, i.e., additional edges (u, v) such that the length of (u, v) is the distance from u to v. Given a weighted, directed graph G and a number $c \in \mathbb{Z}_{>0}$, the *shortcut problem* asks how to insert c shortcuts into G such that the expected number of edges that are contained in an edge-minimal shortest path from a random node s to a random node t is minimal. In this chapter, we study the algorithmic complexity of the problem and give an approximation algorithm for a special graph class. Further, we state ILP-based exact approaches and show how to stochastically evaluate a given shortcut assignment on graphs that are too large to do so exactly.

4.1 Introduction

Background. The problem studied in this chapter originates from the speedup techniques described in the last chapter. One core part of some of these approaches is the insertion of *shortcuts* [BD08, BCD+08, GSSD08, GKW06, GKW07, HSW06, SS06a, SS07, SWW00, SWZ02], i.e., additional edges (u, v) whose length is the distance from u to v and that represent shortest u-v-paths in the graph. The strategies of assigning the shortcuts and of exploiting them during the query differ depending on the speedup technique. Until now, all existing shortcut insertion strategies are heuristics and only few theoretical worst-case or average case results are known [AFGW10, BCK+10a, BCK+10b].

In this context, an interesting new theoretical problem arises: Given a weighted, directed graph G and a number $c \in \mathbb{Z}_{>0}$, the *shortcut problem* asks how to insert c shortcuts into G such that the expected number of edges that are contained in an edge-minimal shortest path from a random node s to a random node t is minimal.

Contribution. In this work we formally state the Shortcut Problem and a variant of it, the Reverse Shortcut Problem. While we study the algorithmic complexity of both problems, the algorithmic contribution focuses on the Shortcut Problem. We state exact, ILP-based solution approaches. We further describe an algorithm that gives an approximation guarantee on graphs in which, for each pair s, t of nodes, there is at most one shortest s-t-path. It turns out that this class is highly relevant as in road networks, most shortest paths are unique and only small modifications have to be made to obtain a graph having unique shortest paths. Finally, we show how to stochastically evaluate a given shortcut assignment on graphs that are too large to do so exactly. Besides its relevance as a step towards theoretical results on speedup-techniques, we consider the problem to be interesting and beautiful on its own right.

Related Work. Parts of this chapter have been published in [BDDW09, BDD$^+$10]. There, an additional approximation strategy is proposed. The approach is based on a partition of the nodes and works on graphs with bounded degree for which shortest paths are unique. It gives an $O\left(\lambda \cdot \max\left\{1, |V|^2/(\lambda^2 c)\right\}\right)$-approximation of the optimal solution, where λ is the number of subsets of the underlying partition and c the number of shortcuts to insert. The diploma thesis [Sch09b] experimentally examines heuristic algorithms to find shortcut assignments with high quality, including local search strategies and a betweenness-based approach. Furthermore, the GREEDY-step Algorithm 4.3 is proposed in this thesis. To the best of our knowledge, the problem of finding shortcuts as stated in this work has never been treated before.

Speedup-techniques that incorporate the usage of shortcuts are the following. Given a graph $G = (V, E)$ the multilevel overlay graph technique [SWZ02, Sch05, HSW06, SS07, HSW08, Hol08] uses some centrality measures or separation strategies to choose a set of 'important' nodes V' in the graph and inserts the shortcuts S such that the graph (V', S) is edge-minimal among all graphs (V', E') for which the distances between nodes in V' are as in (V, E). Highway hierarchies [SS05, SS06a] and Reach Based Pruning [Gut04, GKW06, GKW07, GKW09] iteratively sparsificate the graph according to the 'importance' of the nodes. After each sparsification step, nodes v with small in- and out-degree are deleted. Then for each pair of edges (u, v), (v, w) a shortcut (u, w) is inserted if necessary to maintain correct distances in the graph. SHARC-Routing [BD08, BD09, Del09, BDGW10] and Contraction Hierarchies [GSSD08] use a similar strategy.

Overview. This chapter is organized as follows. Section 4.2 introduces basic definitions. The SHORTCUT PROBLEM and the REVERSE SHORTCUT PROBLEM are stated in Section 4.3. Furthermore, results concerning complexity and non-approximability of the problems are given. The remainder of the chapter focuses on the SHORTCUT PROBLEM. Section 4.4 proposes two exact, ILP-based approaches. In Section 4.5 a greedy algorithm is presented that gives an approximation guarantee on graphs in which shortest paths are unique. A probabilistic approach to evaluate a given solution of the SHORTCUT PROBLEM is introduced in Chapter 4.6. The chapter is concluded by a summary and possible future work in Section 4.7.

4.2 Specific Notation

Throughout this chapter, $G = (V, E, len)$ denotes a directed, weighted graph with positive length function $len : E \to \mathbb{R}_{>0}$. Consider a path $P = (x_1, x_2, \ldots, x_k)$. We say P *contains* node u *before* node v if there are numbers i, j with $0 \leq i \leq j \leq k$ such that $u = x_i$ and $v = x_j$.

Given is a sequence $y_1, \ldots, y_k$ for $k \geq 2$. A y_1-y_2-…-y_k-path is a path P from y_1 to y_k such that P contains node y_i before node y_{i+1} for $i = 1, \ldots, k-1$. A shortest y_1-y_2-…-y_k-path is a y_1-y_2-…-y_k-path that is a shortest path from y_1 to y_k. Let

$$P^-(x, y) := \{s \in V \mid \exists \text{ shortest } s\text{-}y\text{-path containing } x\}$$
$$P^+(x, y) := \{t \in V \mid \exists \text{ shortest } x\text{-}t\text{-path containing } y\}$$

denote the sets of start- or end-vertices of shortest paths through x and y. Similarly, let

$$P(x, y) := \{(s, t) \in V \times V \mid \exists \text{ shortest } s\text{-}t\text{-path that contains } x \text{ before } y\}$$

consist of all pairs of nodes, for which a connecting shortest path containing first x and y exists. Finally, let

$$P^{\bowtie}(x,y) := \{u \in V \mid \exists \text{ shortest } x\text{-}y\text{-path that contains } u\}$$

be the set of all nodes that lie on a shortest x-y-path.

We call a graph G *sp-unique* if, for any pair of nodes s and t in G, there is at most one, unique shortest s-t-path in G. Let $P = (x_1, x_2, \ldots, x_k)$ be a path. The *hop-length* $|P|$ of P is $k-1$. Given two nodes s and t, the *hop-distance* $h_G(s,t)$ from s to t is the minimum hop-length of any shortest s-t-path in G and 0 if there is no s-t-path in G or if $s = t$. We abbreviate $h_G(s,t)$ by $h(s,t)$ if the choice of the graph G is clear. We further assume that for each edge (u,v) in G it is $\text{len}(u,v) = \text{dist}(u,v)$. This can easily be assured by deleting edges (u,v) with $\text{len}(u,v) > \text{dist}(u,v)$ in a preprocessing step. This guarantees that, after the insertion of a shortcut (a,b), there is only one edge (a,b) in the graph.

4.3 Problem Statement and Complexity

In this section, we introduce the Shortcut Problem and the Reverse Shortcut Problem. We show that both problems are NP-hard. Moreover, we show that there is no polynomial-time constant-factor approximation algorithm for the Reverse Shortcut Problem and no polynomial-time algorithm that approximates the Shortcut Problem up to an additive constant unless P = NP. Finally, we identify a critical parameter of the Shortcut Problem and discuss some monotonicity properties of the problem.

In the following, we augment a given graph G with *shortcuts*. These are edges (u,v) that are added to G such that $len(u,v) = dist(u,v)$. A set of shortcuts is called a *shortcut assignment*. We repeat the definition of *shortcut assignment* from the last chapter.

Definition (Shortcut Assignment). Consider a graph $G = (V, E, \text{len})$. A *shortcut assignment* for G is a set $E' \subseteq (V \times V) \setminus E$ such that, for any (u,v) in E', it is $\text{dist}(u,v) < \infty$. The notation $G[E']$ abbreviates the graph G with the shortcut assignment E' added, i.e., the graph $(V, E \cup E', \text{len}')$ where $\text{len}' : E \cup E' \to \mathbb{R}_{>0}$ equals $dist(u,v)$ if $(u,v) \in E'$ and equals $len(u,v)$ otherwise.

When working with shortcuts we are interested in the expected number of edges that are contained in an edge-minimal shortest path from a random node s to a random node t. The *gain* of a shortcut assignment E' measures how much this value decreases due to the graph-augmentation with E'.

Definition (Gain). Given a graph $G = (V, E, \text{len})$ and a shortcut assignment E', the *gain* $w_G(E')$ of E' is

$$w_G(E') := \sum_{s,t \in V} h_G(s,t) - \sum_{s,t \in V} h_{G[E']}(s,t) \ .$$

We abbreviate $w_G(E')$ by $w(E')$ in case the choice of the graph G is clear.

We briefly consider an augmented graph $G[E'] = (V, E \cup E', \text{len}')$ and choose nodes s and t uniformly at random. The expected number of edges on an edge-minimal shortest s-t-path is $\frac{1}{|V|^2}\sum_{s,t\in V} h_{G[E']}(s,t)$ when we count pairs s and t with $\operatorname{dist}(s,t)=\infty$ by 0. The term $\sum_{s,t\in V} h_G(s,t)$ does not depend on E' and hence is constant. Consequently, maximizing the gain and minimizing the expected number of edges on edge-minimal shortest-paths are equivalent problems. The Shortcut Problem consists of adding a number c of shortcuts to a graph, such that the gain is maximal.

Problem (Shortcut Problem). Let $G=(V,E,\text{len})$ be a graph and $c\in\mathbb{Z}_{>0}$ be a positive integer. Given an instance (G,c), the Shortcut Problem is to find a shortcut assignment E' with $|E'|\le c$ such that the gain $w_G(E')$ of E' is maximal.

The Reverse Shortcut Problem searches for a shortcut assignment E' of minimum cardinality achieving at least some given gain k. We assure that such a solution exists by stating an upper bound on k. To obtain k, we first compute the number

$$\left|\{(u,v)\in V\times V \mid \operatorname{dist}(u,v)<\infty, u\neq v\}\right| .$$

This is exactly the value of $\sum_{s,t\in V} h_{G[\overline{S}]}(s,t)$ when inserting all possible shortcuts $\overline{S}$ to G. Then we subtract this value from $\sum_{s,t\in V} h_G(s,t)$ to yield a sharp bound on the gain.

Problem (Reverse Shortcut Problem). Let $G=(V,E,\text{len})$ be a graph and $k\in\mathbb{Z}_{>0}$ be less than or equal to $\sum_{s,t\in V} h_G(s,t)-|\{(u,v)\in V\times V \mid \operatorname{dist}(u,v)<\infty, u\neq v\}|$. Given an instance (G,k) the Reverse Shortcut Problem is to find a shortcut assignment E' such that $w_G(E')\ge k$ and such that $|E'|$ is minimal.

As an auxiliary problem to shorten proofs we also consider the Shortcut Decision Problem.

Problem (Shortcut Decision Problem). Let $G=(V,E,\text{len})$ be a graph and $c,k\in\mathbb{Z}_{>0}$ be positive integers. Given an instance (G,c,k), the Shortcut Decision Problem is to decide if there is a shortcut assignment E' for $G=(V,E,\text{len})$ such that $w_G(E')\ge k$ and $|E'|\le c$.

In order to show the complexity of the problems we make transformations from Set Cover (see page 11) and from its optimization variant Min Set Cover.

Problem (Min Set Cover). Given a collection C of subsets of a finite set U, find a set cover C' of (C,U) of minimum cardinality.

Notation (Solution). Given a {Shortcut Problem, Reverse Shortcut Problem, Min Set Cover}-instance I, we denote by $\text{opt}_{\{\text{SP},\text{RSP},\text{MSC}\}}(I)$ an arbitrary (optimal) solution of I of the according problem.

We now show a relationship between Set Cover and the Shortcut Problem.

Lemma 15. Let (C,U,k) be a Set Cover-instance. Then, there is a graph $G=(V,E,\text{len})$ such that there is a set cover C' for (C,U) of cardinality $|C|'\le k$ if, and only if there is a shortcut assignment E' for G of cardinality $|E|'\le k$ and gain $w(E')\ge(2|C|+1)|U|$. Further, the size of G and the time to compute G is polynomial in the size of (C,U). Finally, given a shortcut assignment E' with $w(E')\ge(2|C|+1)|U|$, we can compute a set cover of cardinality at most $|E'|$ in time polynomial in the size of (C,U,k).

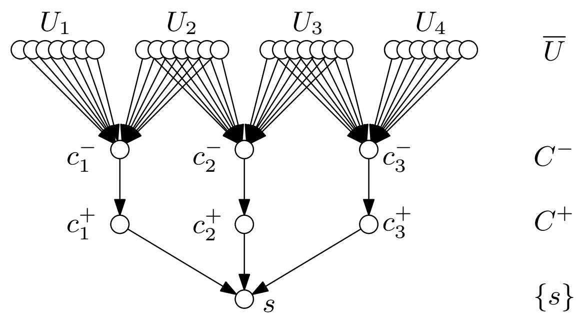


Figure 4.1: Graph $G = (V, E)$ constructed from the SET COVER-instance $\{c_1 = \{1, 2\}, c_2 = \{2, 3\}, c_3 = \{3, 4\}\}$.

Proof. Given an instance (C, U, k) of SET COVER we construct the graph $G = (V, E, \text{len})$ as follows, see Figure 4.1 for an illustration: We denote the value $2|C| + 1$ by Δ. We introduce a node s to G. For each $u \in U$, we introduce a set of nodes $U_u = \{u_1, \dots, u_\Delta\}$ to G. For each c in C, we introduce nodes c^-, c^+ and edges (c^-, c^+), (c^+, s) to G. The graph furthermore contains, for each $u \in U$ and each $c \in C$ with $u \in c$, the edges $(u_r, c^-), r = 1, \dots, \Delta$. All edges are directed and have length 1. We abbreviate $\overline{U} := \bigcup_{u \in U} U_u$, $C^- := \{c^- | c \in C\}$ and $C^+ := \{c^+ | c \in C\}$.

We first observe that shortcuts in G are always contained in one of the following three sets: $\overline{U} \times \{s\}, C^- \times \{s\}$ and $\overline{U} \times C^+$. Given $u \in U$, we say u is *covered* by a shortcut $(c^-, s) \in C^- \times \{s\}$ if $u \in c$.

Claim. Let C' be a set cover of (C, U). Then, the shortcut assignment $E' = \{(c^-, s) \mid c \in C'\}$ fulfills $|E'| = |C'|$ and $w(E') \geq \Delta |U|$.

Obviously $|E'| = |C'|$ holds. For each node $v \in \overline{U}$ the hop-distance to node s decreases by 1 due to the insertion of E'. As $|\overline{U}| = \Delta|U|$ it is $w(E') \geq \Delta|U|$.

Claim. Let E' be a shortcut assignment of G with $w(E') \geq \Delta|U|$. Then, we can construct a shortcut assignment $E'' \subseteq C^- \times \{s\}$ of G with cardinality $|E''| \leq |E|$ and $w(E'') \geq \Delta|U|$ in polynomial time.

We first check if $|E'| > |C|$. In this case we set $E'' := \{(c^-, s) | c \in C\}$ and terminate. Otherwise, we proceed as follows until $E' \subseteq C^- \times \{s\}$ or each $u \in U$ is covered by a shortcut (c^-, s): We choose a shortcut (x, y) in $E' \cap (\overline{U} \times C^+ \cup \overline{U} \times \{s\})$. We further choose a shortcut $(c^-, s) \in V \times V$ such that there is a $u \in c$ which is not covered by any shortcut in E'. Then, we set $E' := (E' \cup \{(c^-, s)\}) \setminus \{(x, y)\}$.

The removal of a shortcut in $\overline{U} \times C^+ \cup \overline{U} \times \{s\}$ decreases the gain by at most 2. Let $u \in U$ be an element that is not covered by a shortcut in E' and let $u \in c \in C$. The insertion of (c^-, s) in E' improves the hop distance $h(v, s)$ for each node in $v \in U_u$ which is not part of a shortcut in E' by 1. As there are $2|C| + 1$ nodes in U_u and we have at most $|C|$ shortcuts, the gain increases by at least $2|C| + 1 - |C|$. Summarizing, at each step $w(E')$ increases at least by $2|C| + 1 - |C| - 2 = |C| - 1 \geq 0$. Any shortcut assignment that covers all $u \in U$ results in the desired gain. Hence, after termination $E'' := E' \cap (C^- \times \{s\})$ gives a solution to the claim.

Claim. Let E' be a shortcut assignment of G with $w(E') \geq \Delta|U|$. Then, we can compute in polynomial time a set cover C' for (C, U) of cardinality at most $|E'|$.

We use the last claim to transform E' such that $E' \subseteq C^- \times \{s\}$ and $w(E') \geq \Delta|U|$. It is $w(E') = |E'| + \Delta|\{u \in U \mid u \text{ is covered by a shortcut in } E'\}| \geq \Delta|U|$. This implies that each $u \in U$ is covered by a shortcut in E' and $\{c | (c^-, s) \in E'\}$ is a set cover of (C, U). $\square$

Theorem 10. The Shortcut Decision Problem is NP-complete.

Proof. Let (C, U, k) be a Set Cover-instance and G be constructed as described in Lemma 15. It is (C, U, k) a yes-instance if and only if the Shortcut Decision Problem-instance $(G, |k|, (|2|C| + 1)|U|)$ is a yes-instance, and the transformation is polynomial. □

We remember that an optimization problem P is NP-hard if there is an NP-hard decision problem P' such that following holds: Problem P' can be solved by a polynomial-time algorithm which uses an oracle that, for any instance of P, returns –in constant time– an optimal solution along with its value.

Corollary. The Shortcut Problem and the Reverse Shortcut Problem are NP-hard.

The transformation applied in Lemma 15 also preserves part of the non-approximability of Min Set Cover.

Theorem 11. Unless P = NP, no polynomial-time constant-factor approximation algorithm exists for the Reverse Shortcut Problem, i.e., there is no combination of an algorithm apx and an *approximation ratio* $\alpha > 0$ such that

- $\mathrm{apx}(G, k)$ is a shortcut assignment for G of gain at least k
- $|\,\mathrm{apx}(G, k)|/|\,\mathrm{opt}_{\mathrm{RSP}}(G, k)| \leq \alpha$ for all instances (G, k) of the Reverse Shortcut Problem
- the runtime of $\mathrm{apx}(G, k)$ is polynomial in the size of (G, k).

Proof. Given a Min Set Cover-instance (C, U), assume to the contrary that there is a polynomial-time constant-factor approximation apx of the Reverse Shortcut Problem with approximation ratio α. Using apx, we construct a constant-factor approximation algorithm for Min Set Cover, contradicting the fact that Min Set Cover is not contained in the class APX unless P = NP [ACG+02]:

As described in Lemma 15, we first construct the graph G. Then we compute $E' = \mathrm{apx}(G, (2|C| + 1)|U|)$ and finally transform E' to a set cover instance C' of (C, U) of size at most $|E'|$. With Lemma 15 we have that

$$|\,\mathrm{opt}_{\mathrm{MSC}}(C, U)| = |\,\mathrm{opt}_{\mathrm{RSP}}(G, (2|C| + 1)|U|)|\ .$$

Hence it is

$$|C'|/|\,\mathrm{opt}_{\mathrm{MSC}}\,| \leq |E'|/|\,\mathrm{opt}_{\mathrm{RSP}}(G, (2|C| + 1)|U|)| \leq \alpha$$

which shows the theorem. □

Theorem 12. Unless P = NP, no polynomial-time algorithm exists that approximates the Shortcut Problem up to an additive constant, i.e., there is no combination of an algorithm apx and a *maximum error* $\alpha \in \mathbb{R}_{>0}$ such that

- $\mathrm{apx}(G, c)$ is a shortcut assignment for G of cardinality at most c
- the runtime of $\mathrm{apx}(G, c)$ is polynomial in the size of (G, c)
- $w_G(\mathrm{opt}_{\mathrm{SP}}(G, c)) - w_G(\mathrm{apx}(G, c)) \leq \alpha$ for all instances (G, c) of the Shortcut Problem.

Proof. Assume to the contrary that there is an polynomial-time algorithm apx that approximates the Shortcut Problem up to an additive constant maximum error α and let $(G = (V, E, \text{len}), c, k)$ be a Shortcut Decision Problem-instance. To assure $\alpha \in \mathbb{Z}^+$, we set $\alpha := \lceil \alpha \rceil$. We construct an instance $(\overline{G} = (\overline{V}, \overline{E}, \overline{\text{len}}), c)$ of the Shortcut Problem by adding to G, for each node $v \in V$, exactly $\chi := \alpha + 1 + |V|^2$ nodes $v_1, \dots, v_\chi$ and directed edges $(v_1, v), \dots, (v_\chi, v)$. We further set $\overline{\text{len}}(v_i, v) = 1$ for $i = 1 \dots \chi$. This construction can be done in polynomial time. Let E' denote $\text{apx}(\overline{G}, c)$.

Our aim is to solve $(G = (V, E, \text{len}), c, k)$ in polynomial time. We can insert at most $c_{\max} := |\{(u, v) \in V \times V \setminus E|\, \text{dist}(u, v) < \infty, u \neq v\}|$ shortcuts into G. If $c \geq c_{\max}$ we can decide the problem in polynomial time by adding all possible shortcuts and computing the according gain. Hence, in the following we may assume $c < c_{\max}$.

Claim. The endpoints of all shortcuts inserted by apx in $\overline{G}$ lie in V, i.e $E' \subseteq V \times V$.

If a shortcut in $\overline{G}$ is not contained in $V \times V$ it must be contained in $\overline{V} \times V$ because of the edge directions in $\overline{G}$. Assume that there is a shortcut $(\overline{u}, v) \in E'$ such that $(\overline{u}, v) \in (\overline{V} \setminus V) \times V$. Removing $(\overline{u}, v)$ from E' will decrease the gain $w_{\overline{G}}(E')$ by at most $|V|^2$ (as it represents only paths starting from $\overline{u}$ of length at most $|V| + 1$). Afterwards inserting an arbitrary shortcut $(x, y) \in V \times V$ increases the gain $w_{\overline{G}}(E' \setminus \{(\overline{u}, v)\})$ by at least χ (as it represents at least χ paths ending at y of length at least 2). Summarizing,

$$w_{\overline{G}}((\{(x, y)\} \cup E') \setminus \{(\overline{u}, v)\}) - w_{\overline{G}}(E') \geq \chi - |V|^2 > \alpha$$

contradicting the approximation guarantee of apx.

Claim. We can use apx to decide $(G = (V, E, \text{len}), c, k)$ in polynomial time contradicting the assumption $\text{P} \neq \text{NP}$.

An exact algorithm can be seen as an approximation algorithm with maximum error $\alpha = 0$. We can show in a similar fashion as in the last claim that an optimal solution of $(\overline{G}, c)$ only consists of shortcuts in $V \times V$, i.e., $\text{opt}_{\text{SP}}(\overline{G}, c) \subseteq V \times V$. Given a shortcut assignment $E'' \in V \times V$ it is $w_{\overline{G}}(E'') = (1 + \chi) \cdot w_G(E'')$. Given an optimal solution E^* for (G, c) and $(\overline{G}, c)$, it follows

$$(1 + \chi) \left(w_G(E^*) - w_G(E')\right) = w_{\overline{G}}(E^*) - w_{\overline{G}}(E') \leq \alpha.$$

Hence, $w_G(E^*) - w_G(E') \leq \alpha/(1 + \chi) < 1$ which implies $w_G(E^*) = w_G(E')$ as both $w_G(E^*)$ and $w_G(E')$ are integer valued. This shows the claim and finishes the proof. □

To obtain a better intuition on the Shortcut Problem, we report some properties of the problem.

Trivial approximation bounds. Consider an arbitrary non-empty shortcut assignment E'. It is $0 \leq \sum_{s,t \in V} h_G(s, t) \leq |V|^3$ for any graph $G = (V, E, \text{len})$ and hence $w_G(E') \leq |V|^3$. As each shortcut in E' decreases the hop-distance from its start to its end-node by at least one 1 we have that each E' is a trivial factor $|V|^3/|E'|$-approximation of the Shortcut Problem. Further, any shortcut assignment achieving the desired gain is a trivial factor $|V|^2$-approximation of the Reverse Shortcut Problem.

Bounded number of shortcuts. If the number of shortcuts we are allowed to insert is bounded by a constant k_{max}, the number of possible solutions of the Shortcut Problem is at most

$$\binom{|V|^2}{k_{max}} = \frac{|V|^2!}{(|V|^2 - k_{max})!k_{max}!} \leq |V|^{2k_{max}}.$$

This is polynomial in the size of the input graph $G = (V, E, \text{len})$. We can evaluate a given shortcut assignment by basically computing all-pairs shortest-paths, hence this can

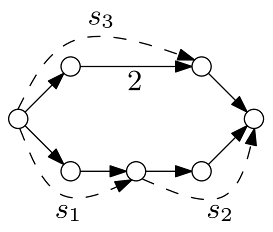


Figure 4.2: Example Graph G with shortcuts s_1, s_2, s_3. All edges for which no weight is given in the picture have weight 1.

be done in time $O(|V|^2 \log |V| + |V||E|)$ using Dijkstra's algorithm. For this reason, the case with bounded number of shortcuts can be solved in polynomial time by a brute-force algorithm.

Monotonicity. In order to show the hardness of working with the problem beyond the complexity results, Figure 4.2 gives an example that, given a shortcut assignment S and an additional shortcut $s \notin S$, the following two inequalities do not hold in general:

$$w(S \cup \{s\}) \geq w(S) + w(s) \tag{4.1}$$

$$w(S \cup \{s\}) \leq w(S) + w(s). \tag{4.2}$$

It is easy to verify that in Figure 4.2 the inequalities $w(\{s_1, s_2\}) > w(s_1) + w(s_2)$ and $w(\{s_1, s_2, s_3\}) < w(\{s_1, s_2\}) + w(s_3)$ hold.

Note that Inequality 4.2 holds if, for any pair of nodes (s, t) of graph G, there is at most one, unique shortest s-t-path in G. We call such a graph *sp-unique* and prove that fact in the following lemma.

Lemma 16. Given an sp-unique graph $G = (V, E, \mathrm{len})$ and a set of shortcuts $S = \{s_1, s_2, \ldots, s_k\}$. Then, $w_G(S) \leq \sum_{i=1}^{k} w_G(s_i)$ and $w_G(S) \leq w_G(\{s_1, \ldots s_{k-1}\}) + w_G(s_k)$.

Proof. Given arbitrary but fixed $a, b \in V$ we denote by $w_G^{ab}(S)$ the gain of S on graph G restricted to shortest a-b-paths, i.e., $w_G^{ab}(S) = h_G(a, b) - h_{G[S]}(a, b)$. Because of $w_G(S) = \sum_{u,v \in V} w_G^{uv}(S)$ it suffices to show $w_G^{ab}(S) \leq w_G^{ab}(\{s_1, \ldots s_{k-1}\}) + w_G^{ab}(s_k)$. The inequality $w_G^{ab}(S) \leq \sum_{i=1}^{k} w_G^{ab}(s_i)$ then follows by induction. We write $s_k = (x, y)$. It is

$$w_G^{ab}(S) = w_G^{ab}(\{s_1, \ldots, s_{k-1}\}) + w_{G[s_1, \ldots, s_{k-1}]}^{ab}(\{(x, y)\}).$$

If $(a, b) \in P(x, y)$ we have

$$w_{G[s_1, \ldots, s_{k-1}]}^{ab}(\{(x, y)\}) \leq h_{G[s_1, \ldots, s_{k-1}]}(\{(x, y)\}) - 1 \leq h_G(\{(x, y)\}) - 1 = w_G^{ab}(s_k).$$

Further, if $(a, b) \notin P(x, y)$ we have $w_{G[s_1, \ldots, s_{k-1}]}^{ab}(\{(x, y)\}) = 0 = w_G^{ab}(s_k)$. Hence, we have

$$w_G^{ab}(S) \leq w_G^{ab}(\{s_1, \ldots s_{k-1}\}) + w_G^{ab}(s_k)$$

which shows the lemma. □

Later, we use these results to present an approximation algorithm for sp-unique graphs.

4.4 ILP-Approaches

In this section we present two exact, ILP-based approaches for the Shortcut Problem. Throughout this section, we are given an instance $(G = (V, E, \text{len}), c)$ of the Shortcut Problem that is to be solved optimally.

For a vertex $x \in V$, we denote by P_x the set of all vertices $u \in V$ for which an x-u-path exists. Remember that we denote by $P^+(x, y)$ the set of all vertices $u \in V$ for which a shortest x-u path containing y exists and that we denote by $P^{\bowtie}(x, y)$ the set of all vertices that lie on a shortest x-y-path. We assume that all distances in the graph are precomputed and hence that the sets P_x, $P^{\bowtie}(x, y)$ and $P^+(x, y)$ are known for all $x, y \in V$.

Simple ILP-Formulation. The following ILP-formulation (SLSP) is straightforward and simple but has the drawback to incorporate $O(|V|^4)$ variables and constraints. The interpretation of the ILP is as follows: The variables $k_t^s(\cdot, \cdot)$ represent an edge-minimal shortest s-t-path in the augmented graph. It is $k_t^s(u, v) = 1$ if and only if the edge (u, v) is used in this path. We characterize all edges or possible shortcuts (u, v) that can be used for a shortest s-t-path by introducing the set

$$A := \{(s, u, v, t) \in V^4 \mid \text{dist}(s, u) + \text{dist}(u, v) + \text{dist}(v, t) = \text{dist}(s, t) < \infty,\ u \neq v\}.$$

Consequently, for fixed $s, v, t \in V$, the set $\{u \in V \mid (s, u, v, t) \in A\}$ contains each node u such that the edge or shortcut (u, v) can be used in a shortest s-t-path. The variable $c(u, v)$ equals 1 if the computed shortcut assignment contains (u, v). Instead of maximizing the gain, our aim is to minimize the sum of all hop-distances in the augmented graph. This value equals the sum of all variables $k_t^s(u, v)$ with $(s, u, v, t \in A)$.

$$\text{(SLSP)} \quad \text{minimize} \sum_{(s,u,v,t)\in A} k_t^s(u, v) \tag{4.3}$$

such that

$$\sum_{\{v\in V \mid (s,v,t,t)\in A\}} k_t^s(v, t) = 1 \qquad s \in V,\ t \in P_s \setminus \{s\} \tag{4.4}$$

$$\sum_{\{u\in V \mid (s,u,v,t)\in A\}} k_t^s(u, v) = \sum_{\{w\in V \mid (s,v,w,t)\in A\}} k_t^s(v, w) \qquad \begin{array}{c} s \in V,\ t \in P_s \setminus \{s\} \\ v \in P^{\bowtie}(s, t),\ v \neq s, t \end{array} \tag{4.5}$$

$$k_t^s(u, v) \leq c(u, v) \qquad (s, u, v, t) \in A,\ (u, v) \notin E \tag{4.6}$$

$$\sum_{(u,v)\in(V\times V)\setminus E} c(u, v) \leq c \tag{4.7}$$

$$k_t^s(u, v) \in \{0, 1\} \qquad (s, u, v, t) \in A \tag{4.8}$$

$$c(u, v) \in \{0, 1\} \qquad (u, v) \in V \times V \setminus E \tag{4.9}$$

Constraint 4.4 and Constraint 4.5 ensure that a shortest path is considered for every s-t-pair: Constraint 4.4 requires that each target t owns exactly one incoming edge on an s-t-path while Constraint 4.5 guarantees that, for each node $v \neq s, t$, there is an incoming edge (on an s-t-path) if there is an outgoing edge (on such a path). The Constraint 4.6 forces shortcuts to be present whenever edges are used that are not present in the graph. Finally, Constraint 4.7 limits the number of shortcuts to be inserted. Consequently, a solution of model (SLSP) gives an optimal solution of (G, c): The set $\{(u, v) \in V \times V | c(u, v) = 1\}$ is a shortcut assignment for G of maximum gain and cardinality at most c.

Obviously, there can be more than one edge-minimal shortest path from a given source to a given target. Hence, the model may incorporate unwanted symmetries. In order to break these symmetries one could use additional constraints. We did not further pursue this direction because of the huge number of constraints that would be necessary. Note that the model stays correct when relaxing Constraint 4.8 to

$$k_t^s(u,v) \in [0,1] \qquad\qquad (s,u,v,t) \in A.$$

Flow-Based ILP-Formulation. The number of variables and constraints of the following integer linear program (LSP) is cubic in $|V|$. The model exhibits two types of variables. It is $c(u, v) = 1$ if and only if the solution found uses the shortcut (u, v). Instead of directly counting the hop-distance for each pair of nodes, we use a flow-like formulation that counts, for each edge, how often it is used in the solution. More detailed, the value of $f^s(u, v)$ can be interpreted as the number of vertices t for which the hop-minimal shortest s-t-path found by (LSP) includes the edge or shortcut (u, v). To characterize all possible combinations of $s, u, v \in V$ such that (u, v) could be an edge or a shortcut in the shortest-paths subgraph with root s, we introduce the set

$$B := \{(s,u,v) \in V^3 \mid \operatorname{dist}(s,u) + \operatorname{dist}(u,v) = dist(s,v) < \infty, \quad u \neq v\} .$$

The flow outgoing from source s is exactly the number of vertices reachable from s (Constraint 4.11). As each node consumes exactly one unit of flow (Constraint 4.12), it is assured that a shortest path from s to any reachable node is considered. Constraint 4.13 forces shortcuts to be present whenever edges are used that are not present in the graph. Finally, Constraint 4.14 limits the number of shortcuts to be inserted. Again, instead of maximizing the gain, our aim is to minimize the sum of all hop-distances in the augmented graph which is given as $\operatorname{obj}(f, c)$.

$$\text{(LSP)} \quad \text{minimize} \quad \operatorname{obj}(f,c) := \sum_{(s,u,v)\in B} f^s(u,v) \tag{4.10}$$

such that

$$\sum_{\{v\in V|(s,s,v)\in B\}} f^s(s,v) = |P_s| - 1 \qquad s \in V \tag{4.11}$$

$$\sum_{\{u\in V|(s,u,v)\in B\}} f^s(u,v) = 1 + \sum_{\{w\in V|(s,v,w)\in B\}} f^s(v,w) \qquad s\in V,\ v\in P_s,\ v\neq s \tag{4.12}$$

$$f^s(u,v) \leq |P^+(s,v)| \cdot c(u,v) \qquad (s,u,v)\in B,\ (u,v)\notin E, \tag{4.13}$$

$$\sum_{(u,v)\in(V\times V)\setminus E} c(u,v) \leq c \tag{4.14}$$

$$f^s(u,v) \in \mathbb{Z}_{\geq 0} \qquad (s,u,v) \in B \tag{4.15}$$

$$c(u,v) \in \{0,1\} \qquad (u,v) \in V\times V \setminus E \tag{4.16}$$

We now prove the correctness of model (LSP). The proof of the following preparatory lemma shows that a solution of (LSP) can be converted into a solution of equal objective value that, for each node, induces a shortest-paths tree.

Lemma 17. There exists an optimal solution (f, c) of (LSP), such that for each $s \in V$, the subgraph induced by $T_s := \{(u, v) \in V \times V \mid f^s(u, v) > 0\}$ is a tree.

Proof. Let (f, c) be a solution of (LSP). Then T_s is a directed acyclic graph with root s as T_s is contained in the shortest-paths subgraph of G with root s. As long as T_s is not a tree proceed as follows:

First, consider an arbitrary node y such that there are two edges (v, y) and (w, y) in T_s. Let x be an arbitrary node such that there are disjoint x-y-paths P_1 and P_2 in T_s. Such a node x has to exist as there is more than one shortest s-y-path in T_s and we can take any topologically maximal node x for which there is more than one x-y-path. Let (y', y) be the last edge on P_1 and $\Delta := f^s(y', y)$. For each edge e on P_1 we set $f^s(e) := f^s(e) - \Delta$, for each edge e on P_2 we set $f^s(e) := f^s(e) + \Delta$.

It is easy to verify that this does not change the feasibility of the solution. Obviously, the objective function cannot decrease because of this operation as (f, c) is optimal. Further, the objective function may not increase: Assume the contrary. Then P_2 contains more edges than P_1. Let (y'', y) be the last edge of P_2 and $\Delta' := f^s(y'', y)$. We would obtain a better feasible solution by setting $f^s(e) := f^s(e) - \Delta'$ for each edge $e \in P_2$ and $f^s(e) := f^s(e) + \Delta'$ for each edge $e \in P_1$, contradicting the optimality of (f, c). □

The following theorem shows that model (LSP) and the Shortcut Problem are equivalent with regard to exact solutions.

Theorem 13. Given an optimal solution E' of the Shortcut Problem, the assignment

$$c'(u,v) := \begin{cases} 1 & ,(u,v) \in E' \\ 0 & ,\text{otherwise} \end{cases}$$

can be extended to an optimal solution of (LSP). Further, given an optimal solution (f, c) of (LSP), the set

$$E'' := \{(u, v) \in V \times V \mid c(u, v) = 1\}$$

is an optimal solution for the Shortcut Problem.

Proof. Let $(G = (V, E, \text{len}), c)$ be a Shortcut Problem-instance. As we have observed before, maximizing the gain is equivalent to finding a shortcut assignment E' that minimizes $\text{obj}(E') := \sum_{s,t \in V} h_{G[E']}(s, t)$. Throughout this proof, we use this point of view.

Let E' be a shortcut assignment of $(G = (V, E, \text{len}), c)$. Consider an arbitrary vertex $s \in V$. There is a shortest-paths tree $T_s \subseteq G[E']$ such that, for each $t \in V$ with $\text{dist}(s, t) < \infty$, the number of edges on the s-t-path in T_s equals $h_{G[E']}(s, t)$. Such a tree T_s can be computed using Dijkstra's algorithm by altering the distance labels to be tuples (edge length, hop distance) and applying lexicographical ordering. Let

$$c'(u,v) = \begin{cases} 1 & ,(u,v) \in E' \\ 0 & ,\text{otherwise} \end{cases}$$

and

$$f'^s(u,v) = \begin{cases} 0 & ,(u,v) \notin T_s \\ |\{w \mid w \text{ is descendant of } v \text{ in } T_s\}| & ,\text{ otherwise} \end{cases}$$

The pair (c', f') is a feasible solution of (LSP). We denote by $P_{T_s}(s, t)$ the s-t-path in T_s and by $|P_{T_s}(s, t)|$ the number of edges on this path. It is

$$\begin{aligned} \sum_{t \in P_s} h_{G[E']}(s,t) &= \sum_{t \in P_s} |P_{T_s}(s,t)| = \sum_{t \in P_s} \sum_{e \in T_s} 1_e(P_{T_s}(s,t)) = \sum_{e \in T_s} \sum_{t \in P_s} 1_e(P_{T_s}(s,t)) \\ &= \sum_{(u,v) \in T_s} |\{w \mid w \text{ is descendant of } v \text{ in } T_s\}| = \sum_{u \in P_s,\ v \in P^+(s,u),\ u \neq v} f'^s(u,v) \end{aligned}$$

Consequently, $\mathrm{obj}(f', c') = \mathrm{obj}(E')$.

On the other hand, let (f, c) be a feasible solution of (LSP). With Lemma 17 we may assume that, for each node s, the subgraph induced by $T_s := \{(u,v) \in V \times V \mid f^s(u,v) > 0\}$ is a tree. Hence, we can show by induction that $f^s(u,v) = |\{w \mid w \text{ is descendant of } v \text{ in } T_s\}|$ for each edge $(u,v) \in T_s$. Further, the set

$$E'' = \{(u,v) \in V \times V \mid c(u,v) = 1\}$$

is a feasible solution of the Shortcut Problem. Finally, we show that $\mathrm{obj}(E'') \leq \mathrm{obj}(f, c)$. We consider each root $s \in V$ separately. To bound the hop-distances in $G[E'']$ starting at s from above we use the shortest-paths tree T_s as a witness. This yields

$$\sum_{t \in P_s} h_{G[E'']}(s,t) \leq \sum_{t \in P_s} |P_{T_s}(s,t)|$$

With the same computation as above, we derive

$$\sum_{t \in P_s} h_{G[E'']}(s,t) \leq \sum_{t \in P_s} |P_{T_s}(s,t)| = \sum_{u \in P_s,\ v \in P^+(s,u),\ u \neq v} f^s(u,v)$$

which shows the claim. □

Tuning the Flow-Based Formulation. In order to simplify model (LSP), we relax Constraint 4.15 to

$$f^s(u,v) \in \mathbb{R}_{\geq 0} \qquad (s,u,v) \in B \tag{4.17}$$

and denote the resulting model (4.10, 4.11, 4.12, 4.13, 4.14, 4.16, 4.17) by (rLSP).

Lemma 18. Let (f, c) be a solution of (rLSP). Then there is a solution (f', c) of (LSP) with same objective value.

Proof. Note that Lemma 17 also holds for (rLSP). Hence, we assume that, for each node s, the subgraph induced by $T_s := \{(u,v) \in V \times V \mid f^s(u,v) > 0\}$ is a tree. The integrality of f now follows by induction on the nodes in reverse topological order and Constraint 4.12. □

In order to heuristically speedup the solving process we may add the following constraints that give bounds on the f-variables.

$$f^s(u,v) \leq |P^+(s,v)| \qquad (s,u,v) \in B \tag{4.18}$$

An additional heuristic improvement works as follows: The sum $\sum_{s,t \in V} h_G(s,t)$ is the value of the objective function of model (LSP) in case no shortcuts are allowed. The value $(h_G(a,b) - 1) \cdot |P(a,b)|$ is an upper bound for the amount that shortcut (a,b) improves the objective function. We precompute $\sum_{s,t \in V} h_G(s,t)$ and, for each pair (a,b) of connected nodes, the value $(h_G(a,b) - 1) \cdot |P(a,b)|$. Then we can add the constraint

$$\underbrace{\sum_{(s,u,v) \in B} f^s(u,v)}_{=\mathrm{obj}(f,c)} \geq \underbrace{\sum_{s,t \in V} h_G(s,t)}_{\text{lower bound of } \mathrm{obj}(f,0)} - \sum_{\substack{a,b \in V \\ \mathrm{dist}(a,b) < \infty}} c(a,b) \cdot \underbrace{(h_G(a,b) - 1) \cdot |P(a,b)|}_{\substack{\text{upper bound of improvement} \\ \text{because of shortcut } (a,b)}} \tag{4.19}$$

to additionally tighten the model.

Case Study. While our main interest on the problem is of theoretical nature, we report some experimental results of the ILP-based approaches. This shall allow for a brief comparison of both formulations and for assessing the heuristic improvements. Our implementation is written in Java using CPLEX 11.2 as ILP-Solver and was compiled with Java 1.6. The tests were executed on one core of an AMD Opteron 6172 Processor, running SUSE Linux 10.3. The machine is clocked at 2.1 GHz and has 16 GB of RAM per processor.

We tested on four different graphs. The graph G_{disk} is a unit-disk graph and generated by randomly assigning 100 nodes to a point in the unit square of the Euclidean plane. Two nodes are connected by an edge if their Euclidean distance is below a given radius. This radius is adjusted such that the resulting graph has approximately 1000 edges. The graph G_{ka} represents a part of the road network of Karlsruhe. It contains 102 nodes and 241 edges. The graph G_{grid} is based on a two-dimensional 10×10 square grid. The nodes of the graph correspond to the crossings in the grid. There is an edge between two nodes if they are neighbors on the grid. Finally, the graph G_{path} is a path consisting of 30 nodes. In each graph, edge weights are randomly chosen integer values between 1 and 1000. For each experiment, the computation time has been limited to 60 minutes. The integrality constraints of the variables $k_s^t(\cdot,\cdot)$ of the simple model and the variables $f^s(\cdot,\cdot)$ of the flow model have been relaxed. Some example outcomes are depicted in Figure 4.3.

The results are summarized in Table 4.1. Columns mean the following: Columns *Eq4.19* and *Eq4.18* indicate if Equation 4.19 and Equation 4.18 are incorporated in the model. For the simple model, we adapted Equation 4.19 in a straightforward fashion. Columns *opt* show if an optimal solution has been found and proven to be optimal. Columns *gap* give the guaranteed approximation ratio of the best feasible solution found within 60 minutes, i.e., the value (*best feasible solution found - best proven lower bound*) / *best proven lower bound.* The value of gap is ∞ if no feasible solution has been found in 60 minutes. Finally, columns *time* give the computation time in minutes.

We observe that the simple model does not benefit from Equation 4.19 and the plain version without this enhancement is always superior. For the flow formulation, it turned out that the version enriched with Equation 4.18 is best: This version is always better than the plain model without improvement and than the formulation enhanced only with Equation 4.19. Further, it is most times better than the version enriched with Equation 4.19 and 4.18. Finally, we see that Equation 4.19 was an improvement to the plain model if more than one shortcut was to be inserted.

Comparing the two formulations we obtain that the flow formulation is superior. The flow formulation enhanced with Equation 4.18 was most times better than the simple model, sometimes with a big gap. With one exception, the difference was small when the simple model was better. Concluding, in this testset the flow formulation enhanced with Equation 4.18 performed best.

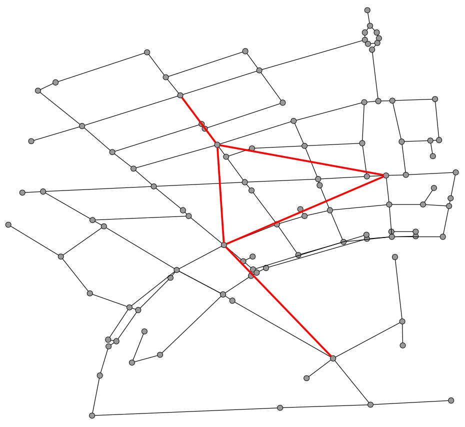
In our experiments we did not take memory consumption into account as the limiting factor was computation time. However, to enable a vague comparison of the memory consumption, we report in Table 4.2 the number of nonzeros reported by CPLEX after the presolve routine. Note that this number turned out to be almost independent from the number of shortcuts to be inserted.

					G_{grid}			G_{ka}			G_{path}			G_{disk}	
shortcuts	model	Eq4.19	Eq4.18	opt	gap	time	opt	gap	time	opt	gap	time	opt	gap	time
1	flow			✓	0	2	✓	0	5	✓	0	1	✓	0	2
1	flow		✓	✓	0	2	✓	0	3	✓	0	0	✓	0	1
1	flow	✓		✓	0	4	✓	0	8	✓	0	0	✓	0	3
1	flow	✓	✓	✓	0	2	✓	0	7	✓	0	0	✓	0	2
1	simple			✓	0	16	✓	0	29	✓	0	1	✓	0	14
1	simple	✓		✓	0	18	✓	0	49	✓	0	1	✓	0	24
2	flow				0.02	60		0.09	60		0.2	60	✓	0	12
2	flow		✓	✓	0	10	✓	0	35	✓	0	8	✓	0	2
2	flow	✓		✓	0	17		0.01	60		0.06	60	✓	0	2
2	flow	✓	✓	✓	0	3	✓	0	40	✓	0	9	✓	0	2
2	simple			✓	0	20	✓	0	26	✓	0	2	✓	0	12
2	simple	✓		✓	0	21	✓	0	48	✓	0	2	✓	0	20
5	flow				0.16	60		0.53	60		0.4	60		0.06	60
5	flow		✓	✓	0	28	✓	0	46		0.16	60	✓	0	4
5	flow	✓			0.05	60		0.12	60		0.39	60	✓	0	55
5	flow	✓	✓		0	60		0.01	60		0.17	60	✓	0	9
5	simple			✓	0	30	✓	0	40		0.04	60	✓	0	15
5	simple	✓		✓	0	58		∞	60		∞	60	✓	0	38
10	flow				0.58	60		0.83	60		0.45	60		0.11	60
10	flow		✓		0.03	60		0.49	60		0.27	60	✓	0	27
10	flow	✓			0.14	60		0.49	60		0.49	60		0.07	60
10	flow	✓	✓		0.05	60		0.34	60		0.32	60	✓	0	25
10	simple				∞	60		∞	60		0.47	60	✓	0	22
10	simple	✓			∞	60		∞	60		2.08	60	✓	0	39

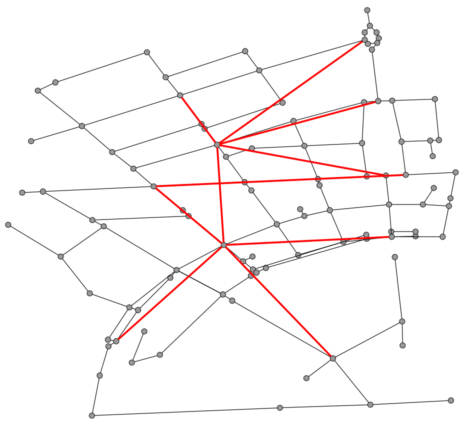
Table 4.1: Experimental results of the ILP-approaches.

model	Eq4.19	Eq4.18	G_{grid}	G_{ka}	G_{path}	G_{disk}
flow			274.818	328.102	34.391	249.564
flow		✓	327.022	392.422	41.849	295.460
flow	✓		342.689	409.157	43.029	311.547
flow	✓	✓	394.737	473.390	50.379	357.146
simple			1.241.560	1.724.034	259.211	1.005.390
simple	✓		1.551.052	2.165.022	324.256	1.250.583

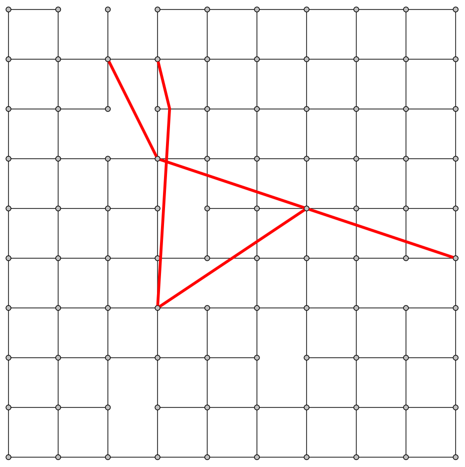
Table 4.2: Number of nonzeros reported by CPLEX after the presolve routine for each model and graph.



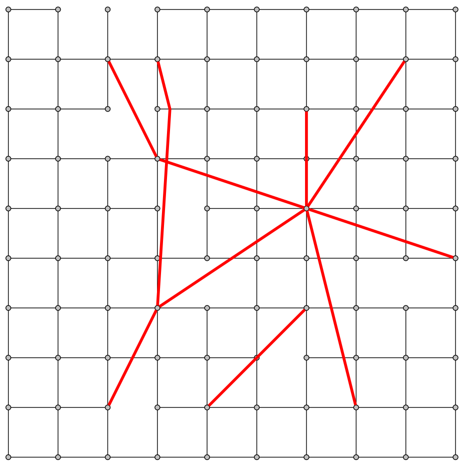
graph *ka* with 5 optimal shortcuts



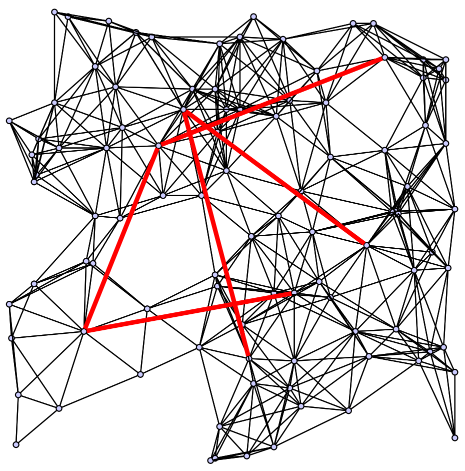
graph *ka* with 10 optimal shortcuts



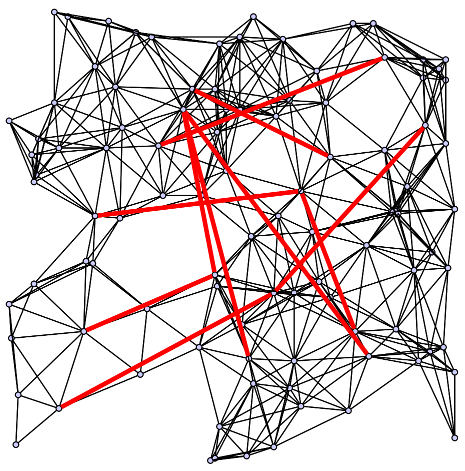
graph *grid* with 5 optimal shortcuts



graph *grid* with 10 optimal shortcuts



graph *disk* with 5 optimal shortcuts



graph *disk* with 10 optimal shortcuts

Figure 4.3: Optimal shortcut assignments for some example graphs.

4.5 Approximation using the Greedy-Strategy

In this section, we propose a polynomial-time algorithm that approximatively solves the Shortcut Problem in a greedy fashion. Given the number c of shortcuts to insert, the approach finds a c-approximation of the optimal solution if the underlying graph is sp-unique. While the algorithm works on arbitrary graphs, we restrict our description to strongly connected graphs to improve readability. The restriction to sp-unique graphs is only needed for achieving the approximation guarantee.

Description. Given the instance (G, c), the Greedy approximation scheme consists of iteratively constructing a sequence $G = G_0, G_1, \dots, G_c$ of graphs where G_{i+1} results from solving the Shortcut Problem on G_i with only one shortcut allowed to insert. The pseudocode for the approach is given as Algorithm 4.1. The following theorem shows the approximation ratio for Greedy.

Algorithm 4.1: GREEDY(G, c)

```
  input : graph $G = (V, E, \text{len})$, number of shortcuts $c$
  output: shortcut assignment $E'$
1 $E' \leftarrow \emptyset$; for $i = 1, 2, \dots, c$ do
2     $(x, y) \leftarrow \arg\max_{(x,y)\in(V\times V)\setminus(E\cup E'),\ \text{dist}(x,y)<\infty}\{w_{G[E']}(\{(x,y)\})\}$
3     $E' \leftarrow E' \cup \{(x, y)\}$
4 output $E'$.
```

Theorem 14. Consider an sp-unique graph $G = (V, E, \text{len})$ together with a positive integer $c \in \mathbb{Z}_{>0}$. The solution $E' := \text{Greedy}(G, c)$ of the Greedy-approach is a c-approximation of an optimal solution E^*, i.e., $w_G(E^*)/w_G(E') \leq c$.

Proof. Let $e_1 \in E'$ be the first shortcut inserted by Greedy. Then, $w_G(e) \leq w_G(e_1)$ for each $e \in E^*$. Moreover by Lemma 16, $w(E^*) \leq \sum_{e\in E^*} w(e)$. This yields

$$w_G(E^*) \leq \sum_{e\in E^*} w_G(e) \leq \sum_{i=1}^{c} w_G(e_1) = c \cdot w_G(e_1) \leq c \cdot w_G(E')$$

which shows $w(E^*)/w(E') \leq c$. □

Basic Runtime Issues. The runtime of Greedy crucially depends on how the next shortcut to be inserted is found. A straightforward approach would be to first precompute the distance $\text{dist}(s, t)$ for each pair $s, t \in V$ as well as the shortest-paths subgraph G_s for each node $s \in V$. Then, the evaluation of a possible shortcut can be done by running breadth-first searches on the $|V|$ graphs G_s. After insertion of a shortcut (a, b) to G, the shortest-paths subgraphs G_s can be adapted by adding (a, b) to each subgraph G_s for which $\text{dist}(s, a) + \text{dist}(a, b) = \text{dist}(s, b)$. Hence G_s contains at most $|E| + c$ edges and the time needed for evaluating one shortcut is $O(|V| \cdot (|V| + |E| + c))$. This leads to a runtime in $O(|V|^2 \cdot |V| \cdot (|V| + |E| + c))$ for evaluating all $|V|^2$ possible shortcuts. The runtime $O(|V|^2 \log |V| + |V| \cdot |E|)$ of precomputing the shortest-paths subgraphs can be neglected.

In the remainder of this section, we show how to perform this step in time $O(|V|^3)$ using a dynamic program. Consequently, the Greedy-strategy can be implemented to work in time $O(c \cdot |V|^3)$.

Greedily finding one optimal shortcut in sp-unique graphs. In sp-unique graphs each shortest path is edge-minimal. Hence, we can compute the gain of a shortcut (a, b) restricted to a pair of nodes $(s, t) \in P(a, b)$ by the equation

$$h_G(s,t) - h_{G[(a,b)]}(s,t) = h_G(a,b) - 1. \tag{4.20}$$

Exploiting this we obtain

$$w(a,b) = (h_G(a,b) - 1) \cdot |P(a,b)| = (h_G(a,b) - 1) \cdot \sum_{s \in P^-(a,b)} |P^+(s,b)|. \tag{4.21}$$

This equation directly leads to Algorithm 4.2 that finds one optimal shortcut for sp-unique graphs. The runtime of the algorithm lies in $O(|V|^3)$ as the computation of $|P^+(s,b)|$ is linear in $|V|$: For each $v \in V$ we have to check if $\text{dist}(s,b) + \text{dist}(b,v) = \mathit{dist}(s,v)$.

Algorithm 4.2: GREEDY STEP ON SP-UNIQUE GRAPHS

```
  input : graph $G = (V, E, \text{len})$, distance table $\text{dist}(\cdot,\cdot)$ of $G$
  output: optimal shortcut $(a, b)$
1 initialize $w(\cdot,\cdot) \equiv 0$
2 compute $h_G(\cdot,\cdot)$
3 for $s \in V$ do
4     for $b \in V$ do
5         compute $|P^+(s,b)|$
6         for $a \in V$ do
7             if $\text{dist}(s,a) + \text{dist}(a,b) = \text{dist}(s,b)$ then
8                 $w(a,b) \leftarrow w(a,b) + (h_G(a,b) - 1)|P^+(s,b)|$
9 output arbitrary $(a, b)$ with maximum $w(a, b)$
```

The problem of this approach is that we can not apply Algorithm 4.2 for the GREEDY-strategy, even when the input graph is sp-unique: After insertion of the first shortcut, the augmented graph is not sp-unique any more and hence we can not use Equation 4.20.

An $O(|V|^3)$-implementation for greedily finding one optimal shortcut. In the following we generalize the above approach to work with arbitrary graphs. The *offset*

$$\omega_{sb}(t) := h_G(s,b) + h_G(b,t) - h_G(s,t)$$

reflects the increase of the hop-distance between given nodes s and t, if we restrict ourselves to shortest paths containing b. We define the *potential gain* $g_s(a, b)$ of a shortcut from a to b with respect to s as

$$g_s(a,b) := h_G(a,b) - 1 - \omega_{sa}(b) \ .$$

This is an upper bound for the decrease of the hop-distance between s and any t in the graph $G[(a, b)]$.

Lemma 19. For all vertices $s, t, a, b \in V$ such that $(s, t) \in P(a, b)$ it holds that

$$h_G(s,t) - h_{G[(a,b)]}(s,t) = \max\{g_s(a,b) - \omega_{sb}(t),\ 0\}.$$

Proof. Directly from the definition of potential gain and offset we obtain

$$g_s(a,b) - \omega_{sb}(t) > 0 \Longleftrightarrow h_G(s,t) > h_G(s,a) + 1 + h_G(b,t) \tag{4.22}$$

Case $[g_s(a,b) - \omega_{sb}(t) > 0]$. Then $h_G(s,t) > h_G(s,a) + 1 + h_G(b,t)$. The presence of shortcut (a,b) decreases the s-t-hop-distance to $h_{G[(a,b)]}(s,t) = h_G(s,a) + 1 + h_G(b,t)$ as the lemma assumes that there is a shortest s-a-b-t-path. This yields

$$\begin{aligned}
h_G(s,t) - h_{G[(a,b)]}(s,t) &= h_G(s,t) - h_G(s,a) - 1 - h_G(b,t) \\
&= h_G(a,b) - 1 \\
&\quad \underbrace{-h_G(s,a) - h_G(a,b) + h_G(s,b)}_{=-\omega_{sa}(b)} \underbrace{-h_G(s,b) - h_G(b,t) + h_G(s,t)}_{=-\omega_{sb}(t)} \\
&= g_s(a,b) - \omega_{sb}(t).
\end{aligned}$$

Case $[g_s(a,b) - \omega_{sb}(t) \le 0]$. With Equation (4.22) we obtain $h_G(s,t) \le h_G(s,a) + 1 + h_G(b,t)$. Hence, a shortcut (a,b) does not improve the hop-distance from s to t and we have $h_G(s,t) - h_{G[(a,b)]}(s,t) = 0$. □

Lemma 19 implies that vertices t in $P^+(s,b)$ with the same value of $\omega_{sb}(t)$ benefit from a shortcut ending at b to the same extent, if we restrict ourselves to shortest paths starting at s. We divide the vertices in $P^+(s,b)$ in equivalence classes with respect to ω_{sb}. Let

$$\Delta_i(s,b) := |\{t \in P^+(s,b) \mid \omega_{sb}(t) = i\}|$$

be the number of vertices in these equivalence classes.

The algorithm we propose makes use of partial (weighted) sums of the $\Delta_i(s,b)$ for fixed s and b in V. For convenience, we introduce two further abbreviations :

$$\begin{aligned}
C_r(s,b) &:= \sum_{i=0}^{r} \Delta_i(s,b) \\
D_r(s,b) &:= \sum_{i=0}^{r} i \cdot \Delta_i(s,b).
\end{aligned}$$

With these definitions, we can form an alternative equation for $w(a,b)$.

Lemma 20. Let $a,b,s,t \in V$ be arbitrary nodes. Then

$$w(a,b) = \sum_{\substack{s \in P^-(a,b) \\ g_s(a,b)>0}} \Big(g_s(a,b) \cdot C_{g_s(a,b)-1}(s,b) - D_{g_s(a,b)-1}(s,b) \Big).$$

Proof.

$$\begin{aligned}
w(a,b) &= \sum_{s,t \in V} \big(h_G(s,t) - h_{G[(a,b)]}(s,t) \big) \\
&= \sum_{(s,t) \in P(a,b)} \big(h_G(s,t) - h_{G[(a,b)]}(s,t) \big) + \sum_{(s,t) \notin P(a,b)} \underbrace{\big(h_G(s,t) - h_{G[(a,b)]}(s,t) \big)}_{=0} \\
&= \sum_{\substack{(s,t) \in P(a,b) \\ \omega_{sb}(t) < g_s(a,b)}} \big(h_G(s,t) - h_{G[(a,b)]}(s,t) \big) + \sum_{\substack{(s,t) \in P(a,b) \\ \omega_{sb}(t) \ge g_s(a,b)}} \underbrace{\big(h_G(s,t) - h_{G[(a,b)]}(s,t) \big)}_{=0 \text{ with Lemma 19}}.
\end{aligned}$$

With Lemma 19, we yield

$$w(a,b) = \sum_{\substack{(s,t)\in P(a,b)\\ \omega_{sb}(t)<g_s(a,b)}} g_s(a,b) - \omega_{sb}(t).$$

It is $\omega_{sb}(t) \geq 0$ as $(s,t) \in P(a,b)$ and hence we have

$$w(a,b) = \sum_{\substack{s\in P^-(a,b)\\ g_s(a,b)>0}} \sum_{i=0}^{g_s(a,b)-1} \sum_{\substack{t\in P^+(s,b)\\ \omega_{sb}(t)=i}} g_s(a,b) - i.$$

As $g_s(a,b)$ is independent of t we can transform the equation as follows

$$\begin{aligned} w(a,b) &= \sum_{\substack{s\in P^-(a,b)\\ g_s(a,b)>0}} \sum_{i=0}^{g_s(a,b)-1} \Delta_i(s,b) \cdot \big(g_s(a,b) - i\big) \\ &= \sum_{\substack{s\in P^-(a,b)\\ g_s(a,b)>0}} \Big(g_s(a,b) \sum_{i=0}^{g_s(a,b)-1} \Delta_i(s,b) - \sum_{i=0}^{g_s(a,b)-1} \big(i \cdot \Delta_i(s,b)\big)\Big) \\ &= \sum_{\substack{s\in P^-(a,b)\\ g_s(a,b)>0}} \Big(g_s(a,b) \cdot C_{g_s(a,b)-1}(s,b) - D_{g_s(a,b)-1}(s,b)\Big). \end{aligned}$$

This finishes the proof. □

Lemma 20 is the key for obtaining our $O(|V|^3)$-algorithm for performing one GREEDY-step, which is stated as Algorithm 4.3: First, all distances and hop-distances are precomputed. We then consider, for each $s \in V$, each shortest-paths subgraph with root s separately. It is easy to see that the values of $\Delta_\cdot(s,\cdot)$, $C_\cdot(s,\cdot)$ and $D_\cdot(s,\cdot)$ can be computed in time $O(|V|^2)$.

Prepared with these values we are ready to apply Lemma 20. We initialize the values $w(\cdot,\cdot)$ with 0. For each triple $s,a,b \in V$, we check if there is a shortest s-a-b-path and if $g_s(a,b) > 0$. We increment $w(a,b)$ according to Lemma 20 in case of a positive answer. Finally, we take an arbitrary shortcut (a,b) that maximizes $w(a,b)$. The correctness of the algorithm directly follows from the definitions of $\Delta_\cdot(\cdot,\cdot)$, $C_\cdot(\cdot,\cdot)$ and $D_\cdot(\cdot,\cdot)$ and Lemma 20. To reach the runtime in $O(|V|^3)$ we answer the question if a shortest s-a-b path exists by checking if $\mathrm{dist}(s,a) + \mathrm{dist}(a,b) = \mathrm{dist}(s,b)$.

Algorithm 4.3: Greedy Step

```
Input: Strongly connected graph $G = (V, E, \text{len})$
Output: shortcut $(a, b)$ maximizing $w_G(\{(a, b)\})$
1  compute $\text{dist}(\cdot,\cdot)$, $h(\cdot,\cdot)$
2  initialize $w(\cdot,\cdot) \equiv 0$
3  initialize $\Delta_i(\cdot,\cdot) \equiv 0$
4  for $s \in V$ do
5      for $b, t \in V$ do                                        /* compute $\Delta$ */
6          if there exists a shortest $s$-$t$-path containing $b$ in $G$ then
7              $j \leftarrow \omega_{sb}(t)$
8              $\Delta_j(s, b) \leftarrow \Delta_j(s, b) + 1$
9      for $b \in V$ do                                           /* compute $C$ and $D$ */
10         $C_0(s, b) \leftarrow \Delta_0(s, b)$
11         $D_0(s, b) \leftarrow 0$
12         for $r := 1$ to $|V| - 1$ do
13             $C_r(s, b) \leftarrow C_{r-1}(s, b) + \Delta_r(s, b)$
14             $D_r(s, b) \leftarrow D_{r-1}(s, b) + r \cdot \Delta_r(s, b)$
15     for $a, b \in V$ do                                        /* apply Lemma 20 */
16         if there exists a shortest $s$-$b$-path containing $a$ and $g_s(a, b) > 0$ then
17             $w(a, b) \leftarrow w(a, b) + g_s(a, b) \cdot C_{g_s(a,b)-1}(s, b) - D_{g_s(a,b)-1}(s, b)$
18 output arbitrary $(a, b)$ with maximum $w(a, b)$
```

4.6 Evaluation of the Measure Function

To evaluate the gain of a given shortcut assignment, a straightforward algorithm requires computing all-pairs shortest-paths. Since this computation is expensive, we provide a probabilistic method to quickly assess the quality of a shortcut assignment E'. This approach can be used for networks where the computation of all-pairs shortest-paths is prohibitive, such as big road networks. For the sake of simplicity we state the approach for the evaluation of $\mu(E') := \sum_{s,t \in V} h_{G[E']}(s, t)$, the adaption to the Shortcut Problem is straightforward. More concrete, we apply the sampling technique to obtain an unbiased estimate for $\mu(E')$ and apply *Hoeffding's Bound* [Hoe63] to get a confidence intervall for the outcome. As an auxiliary result we propose algorithms that approximate the maximum hop-distance in a graph.

Theorem 15 (Hoeffding's Bound). If $X_1, X_2, \ldots, X_K$ are real valued independent random variables with $a_i \le X_i \le b_i$ and expected mean $\mu = \mathbb{E}[\sum X_i / K]$, then

$$\mathbb{P}\left\{\left|\frac{\sum_{i=1}^{K} X_i}{K} - \mu\right| \ge \xi\right\} \le 2e^{-2K^2\xi^2 / \sum_{i=1}^{K}(b_i - a_i)^2}$$

for each $\xi > 0$.

We now model the assessment of a shortcut assignment E' of a graph G in terms of Hoeffding's Bound. Let $X_1, \ldots, X_K$ be the family of random variables such that X_i is defined by

$$X_i := |V| \sum_{t \in V} h_{G[E']}(s_i, t)$$

where $s_i \in V$ is a vertex chosen uniformly at random. We estimate $\mu(E')$ by

$$\hat{\mu} := \sum_{i=1}^{K} X_i/K \ .$$

Because of

$$\mathbb{E}(\hat{\mu}) = \mathbb{E}\left(\sum_{i=1}^{K} \frac{X_i}{K}\right) = \sum_{i=1}^{K} \frac{\mathbb{E}(X_i)}{K} = \mathbb{E}(X_1) = \frac{1}{|V|}\sum_{s\in V}|V|\sum_{t\in V} h_{G[E']}(s,t) = \mu(E')$$

we can apply Hoeffding's Bound if we know lower and upper bounds for the variables X_i. The values 0 and $|V|^3$ are trivial such bounds. We introduce the notion of *shortest-paths diameter* to obtain stronger upper bounds.

Definition. The *shortest-paths diameter* $\mathrm{spDiam}(G)$ of a graph G is the maximum hop-distance from any node to any other node in G.

Applying Hoeffding's Bound with $0 \leq X_i \leq |V|^2\,\mathrm{spDiam}(G)$ yields

$$\mathbb{P}\left\{\left|\hat{\mu} - \mu(E')\right| \geq \xi\right\} \leq 2e^{-2K\xi^2/(|V|^4\cdot \mathrm{spDiam}(G)^2)}$$

and with $l_{rel} := \xi/\hat{\mu}$ we have

$$\mathbb{P}\left\{\left|\frac{\hat{\mu} - \mu(E')}{\hat{\mu}}\right| \geq l_{rel}\right\} \leq 2e^{-2K(\hat{\mu}\cdot l_{rel})^2/(|V|^4\cdot \mathrm{spDiam}(G)^2)}$$

where the parameter l_{rel} states the relative size of the confidence intervall. The values of the variables X_i are chosen by randomly choosing values from the *finite population* $c_1,\dots,c_{|V|}$ *with replacement* where $c_i := |V|\sum_{t\in V} h_{G[E']}(v_i,t)$ and $V = \{v_1,\dots v_{|V|}\}$. In [Hoe63] it is reported that Hoeffding's Bound stays correct if, when sampling from a finite population, the samples are being chosen without replacement. Algorithm 4.4 is an approximation algorithm that exploits the above inequality and that samples without replacement.

Algorithm 4.4: STOCHASTICALLY ASSESS SHORTCUT ASSIGNMENT

```
input : graph $G = (V, E \cup E', \mathrm{len})$,
        size of confidence intervall $l_{rel}$, significance level $\alpha$
output: approximation $\hat{\mu}$ for $\mu = \sum_{s,t\in V} h_G(s,t)$
1 compute random order $v_1, v_2, \dots, v_n$ of $V$
2 compute upper bound $\overline{\mathrm{spDiam}}$ for shortest-paths diameter
3 $i \leftarrow 0$; sum $\leftarrow 0$; $\hat{\mu} \leftarrow 0$
4 while not ($i = |V|$ or $2\cdot\exp(-2i(\hat{\mu}\cdot l_{rel})^2/(|V|^4\overline{\mathrm{spDiam}}(G)^2)) \leq \alpha$) do
5     $i \leftarrow i + 1$
6     $T \leftarrow$ grow shortest-paths tree rooted at $v_i$ (favor edge-minimal shortest paths)
7     sum $\leftarrow$ sum $+|V|\cdot\sum_{t\in V} h'_G(v_i,t)$
8     $\hat{\mu} \leftarrow$ sum $/i$
9 output $\hat{\mu}$
```

Approximating the Shortest-Paths Diameter. A straightforward approach to compute the exact shortest-paths diameter requires computing all-pairs shortest-paths. This is reasonable when working with mid-size graphs that allow the computation of all-pairs shortest-paths at least once and for which a large number of shortcut assignments is to be evaluated. In case the computation of all-pairs shortest-paths is prohibitive one can also use upper bounds for the shortest-paths diameter. We obtain an upper bound the following way:

First we compute an upper bound $\overline{\mathrm{diam}}(G)$ for the diameter of G. To do so we choose a set of nodes $s_1, s_2, \dots, s_l$ uniformly at random. We denote by $\epsilon_G(v) = \max\{\mathrm{dist}_G(v,t) \mid t \in V\}$ the eccentricity of node v in graph $G = (V, E, \mathrm{len})$. For each node s_i, the value $\epsilon_{\overleftarrow{G}}(s_i) + \epsilon_G(s_i)$ is an upper bound for the diameter of G: Let $u, v \in V$ be such that $\mathrm{dist}(u,v) = \mathrm{diam}(G)$. Then

$$\mathrm{diam}(G) = \mathrm{dist}(u,v) \leq \mathrm{dist}(u,s_i) + \mathrm{dist}(s_i,v) \leq \epsilon_{\overleftarrow{G}}(s_i) + \epsilon_G(s_i) \ .$$

We set $\overline{\mathrm{diam}}(G)$ to be the minimum of these values over all s_i. The bound $\overline{\mathrm{diam}}(G)$ is a 2-approximation for the exact diameter $\mathrm{diam}(G)$ of G (already for $l = 1$) as there are $u, v \in V$ and $s_i \in V$ such that

$$\overline{\mathrm{diam}}(G) = \mathrm{dist}(u,s_i) + \mathrm{dist}(s_i,v) \leq \mathrm{diam}(G) + \mathrm{diam}(G) = 2 \cdot \mathrm{diam}(G).$$

Let $\mathrm{len}_{\max}$ and $\mathrm{len}_{\min}$ denote the lengths of a longest and a shortest edge in G, respectively. The value $\overline{\mathrm{diam}}(G)/\mathrm{len}_{\min}$ is an upper bound for $\mathrm{spDiam}(G)$: Let P be an edge-minimal shortest path in G with $|P| = \mathrm{spDiam}(G)$ edges. Then

$$\mathrm{spDiam}(G) = |P| \leq \frac{\mathrm{len}(P)}{\mathrm{len}_{\min}} \leq \frac{\mathrm{diam}(G)}{\mathrm{len}_{\min}} \leq \frac{\overline{\mathrm{diam}}(G)}{\mathrm{len}_{\min}} \ .$$

Further, $\overline{\mathrm{diam}}(G)/\mathrm{len}_{\min}$ is a $2 \cdot \mathrm{len}_{\max}/\mathrm{len}_{\min}$-approximation for $\mathrm{spDiam}(G)$ as with $\mathrm{spDiam}(G) \geq \mathrm{diam}(G)/\mathrm{len}_{\max}$ follows that

$$\frac{\overline{\mathrm{diam}}(G)}{\mathrm{len}_{\min}} \leq \frac{2\,\mathrm{diam}(G)}{\mathrm{len}_{\min}} \leq \frac{2\,\mathrm{len}_{\max} \cdot \mathrm{spDiam}(G)}{\mathrm{len}_{\min}} \ .$$

A more expensive approach works as follows, pseudocode is given as Algorithm 4.5: After computing $\overline{\mathrm{diam}}(G)$, we choose a tuning parameter η. Then we grow, for each node s in G, a shortest-paths tree whose construction is stopped directly before one vertex with distance greater than $\overline{\mathrm{diam}}(G)/\eta$ is settled. When breaking ties between different shortest paths we favor edge-minimal shortest paths. We denote by $\tau_{\max}$ the maximum number of edges of the shortest paths on any of the trees grown *plus one*. Then $\overline{\mathrm{spDiam}} := \tau_{\max} \cdot \eta$ is an upper bound for the shortest-paths diameter of G: Let $P = (v_1, \dots, v_n)$ be an arbitrary edge-minimal shortest path in G. We can split P in sub-paths

$$P_1 = (v_1, \dots, v_{k_1}),\ P_2 = (v_{k_1}, \dots, v_{k_2}),\ \ \dots\ \ ,\ P_\ell = (v_{k_{\ell-1}}, \dots, v_{k_\ell})$$

such that

$$\mathrm{dist}(v_{k_i}, v_{k_{i+1}}) > \overline{\mathrm{diam}}(G)/\eta \quad \text{and} \quad \mathrm{dist}(v_{k_i}, v_{k_{i+1}-1}) \leq \overline{\mathrm{diam}}(G)/\eta \ .$$

The number ℓ of these subpaths is at most η as $\ell > \eta$ would imply that

$$\mathrm{len}(P) > \frac{\overline{\mathrm{diam}}(G)}{\eta}(\ell - 1) \geq \overline{\mathrm{diam}}(G).$$

It is $|P_i| \leq \tau_{\max}$ which yields $|P| \leq \tau_{\max} \cdot \eta$. As P was arbitrary we have that $\mathrm{spDiam}(G) \leq \tau_{\max} \cdot \eta$. Further $\tau_{\max} \cdot \eta$ is a 2η-approximation and an $\eta(1 + 1/(\tau_{\max} - 1))$-approximation of $\mathrm{spDiam}(G)$: With $\tau_{\max} - 1 \leq \mathrm{spDiam}(G)$ follows that

$$\frac{\tau_{\max} \cdot \eta}{\mathrm{spDiam}(G)} \leq \frac{(\mathrm{spDiam}(G) + 1)\eta}{\mathrm{spDiam}(G)} = \eta(1 + \frac{1}{\mathrm{spDiam}(G)}) \leq \eta(1 + \frac{1}{\tau_{\max} - 1}) \leq 2 \cdot \eta \ .$$

Algorithm 4.5: COMPUTE UPPER BOUND FOR SHORTEST-PATHS DIAMETER

```
input : graph $G = (V, E, \mathrm{len})$, tuning parameter $l$, tuning parameter $\eta$
output: upper bound $\overline{\mathrm{spDiam}}$ for the shortest-paths diameter of $G$
1 $\overline{\mathrm{diam}}(G) \leftarrow \infty$; $\tau \leftarrow 0$;
2 for $i = 1, \ldots, l$ do                                  /* compute $\overline{\mathrm{diam}}(G)$ */
3     $s \leftarrow$ choose node uniformly at random
4     grow shortest-paths tree rooted at $s$
5     grow shortest-paths tree rooted at $s$ on the reverse graph $\overleftarrow{G}$
6     $\overline{\mathrm{diam}}(G) \leftarrow \min\{\overline{\mathrm{diam}}(G), \max_{v \in V}\{\mathrm{dist}(s, v)\} + \max_{v \in V}\{\mathrm{dist}(v, s)\}\}$
7 for $s \in V$ do                                          /* compute $\overline{\mathrm{spDiam}}(G)$ */
8     $T$ $\leftarrow$ grow partial shortest-paths tree rooted at $s$
             (favoring edge-minimal shortest paths). Stop growing the tree directly
             before the first node with $\mathrm{dist}(s, v) > \overline{\mathrm{diam}}(G)/\eta$ is settled.
      $\tau_{\max} \leftarrow \max\{\tau_{\max}, 1 +$ maximal number of edges of a path in $T$ $\}$
9 output $\overline{\mathrm{spDiam}} := \tau_{\max} \cdot \eta$
```

Obviously, the whole proceeding only makes sense for graphs for which the shortest-paths diameter is much smaller than the number of nodes. This holds for a wide range of real-world graphs, in particular for road networks. For example, the road network of Luxembourg provided by the PTV AG [PTV08] consists of 30733 nodes and has a shortest-paths diameter of only 429. The road network of the Netherlands consists of 946.632 nodes and has a shortest-paths diameter of 1503.

4.7 Conclusion

Summary. In this chapter we studied two problems. The SHORTCUT PROBLEM is the problem of how to insert c shortcuts in G such that the expected number of edges that are contained in an edge-minimal shortest path from a random node s to a random node t is minimal. The REVERSE SHORTCUT PROBLEM is the variant of the SHORTCUT PROBLEM where one has to insert a minimal number of shortcuts to reach a desired expected number of edges on edge-minimal shortest paths.

We proved that both problems are NP-hard and that there is no polynomial-time constant-factor approximation algorithm for the REVERSE SHORTCUT PROBLEM, unless P = NP. Furthermore, no polynomial-time algorithm exists that approximates the SHORTCUT PROBLEM up to an additive constant unless P = NP.

The algorithmic contribution focused on the SHORTCUT PROBLEM. We proposed two ILP-based approaches to exactly solve the SHORTCUT PROBLEM: A straightforward formulation that incorporates $O(|V|^4)$ variables and constraints and a more sophisticated flow-like formulation that requires $O(|V|^3)$ variables and constraints. We considered a greedy approach that computes a c-approximation of the optimal solution if the input graph is such that shortest paths are unique. We further presented a dynamic program that performs a greedy step in time $O(|V|^3)$ which yields an overall runtime in $O(c \cdot |V|^3)$. Finally, we proposed a probabilistic method to quickly evaluate the measure function of the Shortcut Problem. This can be used for large input networks where an exact evaluation is prohibitive.

Future Work. A wide range of possible future work exists for the Shortcut Problem. From a theoretical point of view the probably most interesting open field is the approximability of the Shortcut Problem. It is still unknown if it is in APX. Furthermore, it would be helpful to identify graph classes for which the Shortcut Problem or the Reverse Shortcut Problem becomes tractable. FPT-algorithms are also desirable. From an experimental point of view it would be interesting to develop heuristics that find good shortcuts for large real-world inputs. In particular, evolutionary algorithms and local search algorithms seem promising.

Finally, we pose the question if the given ILP-approaches can be used for the design of approximation algorithms. We do not see good chances for rounding-based methods. However, other techniques like primal-dual arguments might work.

Chapter 5

Batch-Dynamic Single-Source Shortest-Paths Algorithms

A dynamic shortest-paths algorithm is called a batch algorithm if it is able to handle graph changes that consist of multiple edge updates at a time. In this chapter we focus on fully-dynamic batch algorithms that compute the distances from one single node to each other node in directed graphs with positive edge weights. We give an extensive experimental study of the existing algorithms for the single-edge and the batch case, including a broad set of test instances. We further present tuned variants of the already existing SWSF-FP-Algorithm being up to 15 times faster than SWSF-FP. A surprising outcome of our experiments is the astonishing level of data dependency of the algorithms.

5.1 Motivation

The *single-source shortest-paths problem* is a fundamental graph problem with many real-world applications, such as routing in road networks, routing/data harvesting in sensor networks and internet routing using link state protocols (for example OSPF and IS-IS). In these applications shortest-paths trees are stored and have to be updated whenever the underlying graph undergoes changes [NST00, BCD$^+$08, DW07, WW07a].

Algorithms that update the trees without a full recomputation from scratch are called *dynamic single-source shortest-paths algorithms.* Such algorithms differ slightly in the type of their output. Some store only the distances from the source, while others additionally store a shortest-paths tree or the shortest-paths subgraph. Some of the algorithms known in the literature are only able to cope with the update of one edge at a time, while others can perform *batch updates*, i.e., update the shortest-paths information after multiple edges have simultaneously changed their weights. Batch updates naturally arise in real-world applications: traffic jams usually affect a set of edges, updating the information in sensor networks usually requires flooding, which is done in intervals big enough so that more than one link in the network may have changed.

We consider edge insertions and deletions as special cases of weight changes: Deletions correspond to weight increments *to* infinity, while insertions are weight decrements *from* infinity. An algorithm is called *fully dynamic* if both weight increases and decreases are supported, and *semi-dynamic* if only weight decreases or only weight increases are supported.

Aims. In this chapter we focus on fully-dynamic batch updates for directed graphs with positive edge weights. In order to compare the different approaches, the only requirement that we make regarding the tested algorithms is that they update the distance vector. Hence, we do not demand that the algorithm's output contains a shortest-paths tree or the shortest-paths subgraph. We furthermore require that the algorithms be able to cope with edge insertions and deletions. For our experimental study, we use integer edge weights.

Up to now, the experimental knowledge on this topic has been quite sparse. The existing experimental work focuses on very specific datasets. Hence, our first interest is to study the performance when applying single-edge updates to graphs with different structural properties. Do algorithms behave uniformly or is there a high degree of data dependency? This question will be answered in an experimental study, including a broad set of real-world and synthetic instances. For batch updates more fundamental open questions exist: It even is unknown if it is useful to process a set of updates as a batch. Intuition tells us that edge updates that are far away from each other do not interfere regarding their impact on the shortest paths in the graph. So it seems that these updates can be handled independently. On the other hand, updated edges with a strong interference should be processed in a batch (paying with some computational overhead). We show that this intuition is right, and how to formalize the interference of updated edges through a simple approach. Finally, we pay some attention to the already existing SWSF-FP-algorithm. This algorithm has been stated with regard to mainly theoretical considerations. We test if it can be implemented more efficiently and if combinations with other algorithms yield additional speedup.

Related work. Part of this work has already been published in [BW09a, BW09b]. Ramalingam and Reps [RR96b] introduce the batch algorithm SWSF-FP, Narváez et al. [NST00] propose the Narváez-framework containing six single-edge update algorithms and a modification to the framework leading to the according batch algorithms. Pure single-edge update algorithms are RR [RR96a] (due to Ramalingam and Reps) and FMN [FMSN00] (by Frigioni et al.). All these algorithms are described shortly in Section 5.3. Buriol et al. [BRT08] present a heuristic technique to speed up RR-like approaches. The technique is similar to techniques used in the Narváez-framework but does not support edge insertions or deletions. Furthermore, in [BRT08] the RR algorithm is adapted to maintain a special (shortest-paths) tree proposed in [KT01].

There is no algorithm known in the literature for which the worst case is asymptotically better than recomputing the new solution from scratch. In the original works the algorithms are theoretically analyzed with respect to different measures. These measures mostly depend on the size of the subgraph for which the shortest-paths subgraph changes. An overview of these complexity results can be found in Appendix B.

There is some work on the variant of the problem where edge weights may also be negative. In [RR96a] the algorithm RR is adapted to cope with the existence of negative cycles, in [FMSN03] the same is done for the algorithm FMN. In [Dem01] Demetrescu gives some algorithms for that problem. These algorithms use the reweighting technique, which incorporates a complete Dijkstra run on the graph (with changed edge weights). Hence, this approach is impractical for the problem with positive edge weights.

A well-studied related problem is the *fully dynamic all-pairs shortest-paths problem*, in which the distances between all pairs of nodes have to be maintained while the graph undergoes changes. See [DI06] for a survey on the problem.

There is only little experimental work on this topic, all concentrating on single-edge updates. In [FINP98] the algorithms FMN, RR and a full recomputation from scratch are compared on two instance classes: Erdős-Rényi graphs, where updates are chosen uniformly at random and a graph representing the internet on the AS-level, where updates simulate the failure and recovery of the links. In [NST00] the algorithms of the

NARVÁEZ-framework are evaluated on graphs originating from a generator. This generator randomly places nodes on a grid and connects them by edges with probability that exponentially decreases with the distance of the nodes. The generator does not seem to be available anymore. In [TTIW07] the algorithms SWSF-FP, RR, FMN, NARVÁEZ and a full recomputation from scratch using DIJKSTRA, BELLMAN FORD and D'ESOPO PAPE are evaluated with single-edge updates on Erdős-Rényi-like graphs. In [DW07] one algorithm of the NARVÁEZ-framework is evaluated on random single-edge updates on a graph representing the road-network of Western Europe. In [BRT08], the algorithm RR as well as seven variants thereof are evaluated on a real world AT&T IP network, synthetic internet-related graphs and a large set of other synthetic instances, namely those of [CGR96] with non-negative edge lengths.

Overview. This chapter is organized as follows. Section 2 states basic definitions and formally introduces the problem. Section 3 reviews the existing algorithms. Section 4 presents our tuned variants of the SWSF-FP-algorithm, while an extensive experimental study of these algorithms on synthetic and real-world data is given in Section 5. The chapter ends with a conclusion in Section 6.

5.2 Problem Statement

Throughout this chapter let $G = (V, E, \mathrm{len})$ be a directed graph with positive length function $\mathrm{len}: E \to \mathbb{R}_{>0} \cup \{\infty\}$. Let $s \in V$ be an arbitrary but fixed *source*.

A *batch update* is a set of *edge modifications* on G which can be edge insertions, edge deletions, edge weight increases and edge weight decreases (that keep the length function positive). We are given a distance vector $D[]$ such that $D[v] = \mathrm{dist}(s, v)$ for each node $v \in V$. We want to maintain $D[]$ in a dynamic environment where G is undergoing batch updates. After each batch update, $D[]$ and possibly required auxiliary data needed by the recomputation algorithm has to be updated accordingly.

Throughout the text we recompute $D[]$ and the auxiliary data when *one* concrete batch update is given. Because of the recomputation of the auxiliary data the algorithms are able to handle subsequent updates. For notational convenience, we consider inserted or deleted edges to be existing in the original and the updated graph and set the edge length to infinity if necessary.

Some of the following algorithms are designed to handle only one edge modification at a time. Obviously, repeated application of these algorithms also solves the batch case. We call such algorithms *iterative algorithms* while the others are called *batch algorithms*. Whenever we use an iterative algorithm, the single-edge updates are processed in a random order. If we compare different iterative algorithms we always apply the same order for each algorithm.

Iterative algorithms can be split into two parts: the *incremental* part handles edge insertions and weight decreases while the *decremental* part handles edge deletions and weight increases. This terminology can be unintuitive on a first glance but originates from the point of view that the graph increases when edges are inserted.

5.3 Description of Algorithms

In this section we describe the algorithm of Ramalingam and Reps, the approach of Frigioni et al. and the NARVÁEZ-framework. The algorithm SWSF-FP and tuned variants of it are described in the next section.

Each description starts with an outline explaining the key ideas of the approach. This part suffices to inform about the behavior and main concepts of the strategy. The outline is followed by a more technical description of the algorithms and data structures applied. The technical description can be used to reconstruct, in more detail, what has been implemented. Pseudocode and more extensive explanations can be found in the original papers. A review on complexity results can be found in Appendix B.

5.3.1 General Approach and Notation.

Setting. The input of the algorithms is the outdated distance vector $D[]$, the graph G, the original length function len, the batch update U and some auxiliary data which is described for each algorithm separately. The output is the updated distance vector $D[]$ and the updated auxiliary data.

Consistency. The *consistent value* $\mathrm{con}(v)$ of a node v is

$$\mathrm{con}(v) := \begin{cases} \min_{(u,v)\in E}\{D[u] + \mathrm{len}(u,v)\} & , \quad v \neq s \\ 0 & , \quad v = s \end{cases}$$

A node is said to be *consistent* if $D[v] = \mathrm{con}(v)$ and to be *overconsistent* if $D[v] > \mathrm{con}(v)$. Further, v is said to be *inconsistent* if v is not consistent. If, for each incoming edge (u, v) of a consistent node v, we have $D[u] = \mathrm{dist}(s, u)$, then it is also $D[v] = \mathrm{dist}(s, v)$. Hence, when all nodes are consistent, the Bellman-Ford Equations are fulfilled and $D[]$ contains the correct distances from the source.

Initialization Phase. Each algorithm contains an initialization phase that updates the edge lengths. Further it assures that all nodes are consistent or overconsistent. Hence, after the initialization phase, $D[v]$ is an upper bound for $\mathrm{dist}(s, v)$.

Main Phase. Each algorithm includes a main phase in which a min-based priority queue Q is used to recompute the distances in a Dijkstra-like fashion but on a hopefully smaller subgraph. As the aim is to make overconsistent nodes consistent, we have to search for edges that cause overconsistency. This is done by *relaxing* edges.

In this chapter, we say we *relax* an edge (u, v) when we check if $D[v] > D[u]+\mathrm{len}(u, v)$. An edge (u, v) is said to be *consistent* if $D[v] = \mathrm{len}(u, v) + D[u]$ and *underconsistent* if $D[v] > \mathrm{len}(u, v)+D[u]$. We use the convention $\min\emptyset := \infty$, i.e., we consider the minimum of the empty set to be infinity.

5.3.2 Algorithm of Ramalingam and Reps

Ramalingam and Reps [RR96a] describe the iterative algorithm RR that handles only edge insertions and deletions. It can directly be transfered to an algorithm that also works with edge weight increases and decreases. We state this variant. The main idea is to store additional information on the shortest-paths subgraph that helps to speedup the recomputation.

Outline. This is an iterative approach, hence updates are handled edge by edge. The algorithm additionally maintains the shortest-paths subgraph $\mathcal{S}$ of G with root s. Further, for each node v, the indegree of v in $\mathcal{S}$ is known.

Given the updated edge (x, y), the incremental part first checks if the distance from s to y decreases because of the update (remember that the incremental part handles edge weight *decreases*). This is done by relaxing the edge (x, y). In that case, the distance label is updated and y is inserted into the priority queue.

Given the updated edge (x, y), the decremental part removes (x, y) from $\mathcal{S}$ if $(x, y) \in \mathcal{S}$. Then, the subgraph $B \subseteq \mathcal{S}$ that is not connected to s any more is identified. For each node $b \in B$ we compute an upper bound for the distance from s as follows: If b does not own an incoming edge from outside B, we set $D[b] := \infty$. Otherwise, we set $D[b] := \min\{D[a] + \text{len}(a, b) \mid (a, b) \in E, a \notin B\}$ and insert b with priority $D[b]$ in the queue. Then, the main phase starts.

The main phase of both, decremental and incremental algorithm behaves like the main phase of Dijkstra's algorithm. The structure of $\mathcal{S}$ and the corresponding indegrees are updated whenever necessary.

The advantages of the approach are the small overhead compared with Dijkstra's algorithm and the easy detection of the possibly affected area in the decremental part: The knowledge on $\mathcal{S}$ helps to compute the set B in the decremental part which can be done by performing an altered depth first search. Note, that the incremental part does not benefit from the knowledge on $\mathcal{S}$.

Auxiliary Data. The approach maintains the following auxiliary data: for each edge (u, v), a binary flag that indicates if (u, v) lies on the shortest-paths subgraph $\mathcal{S}$ rooted at s and for each node v, the number $\text{indeg}(v)$ of incoming edges of v in $\mathcal{S}$. In the following indeg is adjusted whenever $\mathcal{S}$ changes.

Incremental Part. Given the edge (x, y) with weight decrease, we first update $\text{len}(x, y)$ and relax (x, y). If (x, y) is consistent we insert the edge (x, y) in $\mathcal{S}$. If (x, y) is underconsistent we set $D[y] := D[x] + \text{len}(x, y)$ and insert y with priority $D[y]$ into Q.

Main phase. We perform as follows until Q is empty: We extract and delete the minimum node v from Q. Then, each edge with target v is removed from $\mathcal{S}$ and each consistent incoming edge (u, v) is inserted into $\mathcal{S}$. Afterwards, for each outgoing consistent edge (v, w) we insert (v, w) into $\mathcal{S}$. For each outgoing underconsistent edge (v, w) we set $D[w] := D[v] + \text{len}(v, w)$ and insert w with priority $D[w]$ in Q.

Decremental Part. Given the edge (x, y) with weight increase, we first update $\text{len}(x, y)$. If $(x, y) \notin \mathcal{S}$, nothing is to do. Otherwise, we remove (x, y) from $\mathcal{S}$. We now compute the subgraph $B \subseteq \mathcal{S}$ of $\mathcal{S}$ that is not connected with s any more: If $\text{indeg}(y) > 0$ nothing is to do. Otherwise we initialize the set $W := \{y\}$. Then, we iteratively extract elements v from W. Directly after extracting v we check, for each outgoing edge (v, w), if $(v, w) \in \mathcal{S}$. In this case we remove (v, w) from $\mathcal{S}$ and insert w into W if $\text{indegree}(w) = 0$. Afterwards, B is the set of all nodes that have been inserted into W. In order to save overhead for computing both, W and B, we implemented W as a FIFO that never forgets extracted elements.

For each node $b \in B$, we set $D[b] := \min\{D[a] + \text{len}(a, b) \mid (a, b) \in E, a \notin B\}$. As we use the convention $\min \emptyset = \infty$ it is $D[b] = \infty$ if b has no incoming edge from outside B. If $D[b]$ is not infinity we insert b with priority $D[b]$ into Q.

Main phase. Now, we perform as follows until Q is empty: We extract and delete the minimum node v from Q. Each incoming consistent edge (u, v) is inserted into $\mathcal{S}$. For each outgoing underconsistent edge (v, w), we assign $D[w] := D[v] + \text{len}(v, w)$ and insert w with priority $D[w]$ into Q.

5.3.3 Algorithm of Frigioni et al.

The FMN-algorithm of Frigioni et al. [FMSN00] is an iterative algorithm that uses more complex auxiliary data to obtain better theoretical worst case bounds for a special graph class. Its main ingredient are two additional priority queues attached to each node that prevent the algorithm from relaxing some unnecessary edges.

k-Bounded Accounting Functions. The approach relies on the existence of a *k-bounded accounting function* on $G = (V, E, \text{len})$, which is a mapping $K : E \to V$ such that for each edge (u, v) the node $K(u, v)$ is either u or v and such that for each node n, no more than k edges are n-valued. For our experiments, we use the constructive 2-approximation algorithm described in [FMSN03] for finding a k-bounded accounting function on G.

Edge insertions and deletions may change the k-bounded accounting function used within the algorithm. An adaption of the approach coping with that problem is described in [FMSN00].

Auxiliary Data. The algorithm stores a k-bounded accounting function K of the graph. Given a node x, the set of edges e with $K(e) = x$ is called ownership(x). The set of the other edges adjacent to x is called not_ownership(x). For each node x, we store two additional lists that contain all incoming and outgoing edges in ownership(x). Later, these lists are used to quickly iterate over such elements. The *backward level* of an edge (z, q) is the value

$$\text{b\_level}_z(q) := D[q] - \text{len}(z, q).$$

The *forward level* of an edge (z, q) is the value

$$\text{f\_level}_z(q) := D[q] + \text{len}(z, q).$$

For each node x, the algorithm stores two priority queues, each containing the edges in not_ownership(x). The queue B_x is max-based. The priority of an edge (x, y) is $\text{b\_level}_x(y)$. The queue F_x is min-based. The priority of an edge (x, y) is $\text{f\_level}_x(y)$. In the original version a shortest-paths tree is additionally maintained by storing a parent node $P[n]$ for each node $n \neq s$. We do not store that tree.

Roughly speaking, the additional priority queues help to save effort when relaxing edges during the recomputation of the distances. The separation in ownership-edges and not_ownership-edges is used to balance the benefit of the queues and the extra work necessary to keep the information in the queues up-to-date.

Outline. This algorithm basically works like the algorithm of Ramalingam and Reps. There are two differences. Firstly, the shortest-paths subgraph and the corresponding indegrees are not maintained. Hence, the initialization phase of the decremental part can not make use of that information. Secondly, the additional queues attached to each node may help to relax fewer edges:

Consider the main phase of the incremental part. We have just extracted a node v and the next task is to relax all edges outgoing from v. Using the queue B_v we may save some relaxations. We relax the edges (v, w) in not_ownership(v) in descending order of $\text{b\_level}_v(w)$. We can stop after the first edge (v, w) with $\text{b\_level}_v(w) = D[w] - \text{len}(v, w) < D[v]$ is reached. All following edges (v, x) can not effect in improving $D[x]$. We still have to relax all outgoing edges in ownership(v) as these are not contained in B_v.

Now consider the initialization phase of the decremental part. We have to identify the branch of the shortest-paths subgraph $\mathcal{S}$ that is disconnected from s after removing the updated edge (x, y) from $\mathcal{S}$. As the shortest-paths subgraph and the corresponding indegrees are not explicitely stored anymore (as they are in the algorithm RR) we use a straightforward search starting at the updated edge. This search is explained in the technical part. The search needs, when visiting a node v, to check if there is an incoming edge (u, v) that lies on the shortest-paths subgraph $\mathcal{S}$ and whose source node u already has been visited. This could be checked by checking if $D[u] + \text{len}(u, v) = D[v]$. We can use the queue F_v to speed up this part similar to the argument described above for the queue B_v.

Incremental Part. Given the edge (x, y) with weight decrease, we first update $\text{len}(x, y)$ and update the queues $B(x)$, $F(x)$, $B(y)$ and $F(y)$. Then, if (x, y) is underconsistent, we set $D[y] = D[x] + \text{len}(x, y)$ and insert y with priority $D[y]$ into Q.

In the main phase we perform as follows until Q is empty: We extract and delete the minimum node v from Q and update the queues $B(v)$ and $F(v)$ (i.e., we recompute backward and forward level of all edges adjacent to v but not owned by v and re-order the priority queues accordingly). Afterwards we check for each edge (v, w) in ownership(v) and for each edge (v, w) in not_ownership(v) with $\text{b\_level}_v(w) > D[v]$ if (v, w) is underconsistent. In that case we set $D[w] = D[v] + \text{len}(v, w)$ and call $Q.\textsc{InsertOrUpdate}(v, D[v])$.

Decremental Part. Given the edge (x, y) with weight increase, we first update $\text{len}(x, y)$ and the queues $B(x)$, $F(x)$, $B(y)$ and $F(y)$. Let $\mathcal{S}$ be the tentative shortest-paths subgraph that is induced by all consistent edges. Let $B \subseteq \mathcal{S}$ be the subgraph of $\mathcal{S}$ that is not connected to s anymore. We compute B as follows.

We apply another priority queue M and insert y with priority $D[y]$ into M. Then we proceed as follows until M is empty. We extract a node v with minimum priority from M and check for all incoming edges (u, v) if $(u, v) \in \mathcal{S}$ and if $u \notin B$. This is done by checking all edges in ownership(v) in the straightforward way and by considering all edges in F_v ordered by the according priority until an edge (u, v) with $u \notin B$ is found.

If there is no edge $(u, v) \in \mathcal{S}$ with $u \notin B$, we insert v into B and for each outgoing edge (v, w) we insert w with priority $D[w]$ into M if $(v, w) \in \mathcal{S}$ and w is not already contained in M.

For each node $b \in B$, we set $D[b] := \min\{D[a] + \text{len}(a, b) \mid (a, b) \in E, a \notin B\}$ and insert b with priority $D[b]$ into Q if $D[b] < \infty$ (remember that $\min \emptyset = \infty$).

Main phase. Now, we perform as follows until Q is empty: We extract and delete the minimum node v from Q and update the queues $B(v)$ and $F(v)$. For each outgoing underconsistent edge (v, w) we assign $D[w] := D[v] + \text{len}(v, w)$ and call the routine $Q.\textsc{InsertOrUpdate}(w, D[w])$.

Reading the Priority Queues B_v and F_v. In the above algorithm we sometimes have the task to read the k biggest values in a priority queue or to find the biggest value of all elements that fulfill a given property. This could be done by using EXTRACTMIN-operations followed by INSERT-operations to rebuild the original state of the queue. We did not use this approach. As we implemented the priority queues as binary heaps we searched directly inside the binary tree with a pruned depth/breadth first search.

5.3.4 Algorithm of Narváez et al.

Narváez et al. [NST00] propose a batch algorithm incorporating two degrees of freedom. The main idea of the NARVÁEZ-framework is to maintain a tentative shortest-paths tree and to early-propagate distance changes through that tree. Please note that, although we consider the approach to be a batch algorithm, edge updates are handled iteratively in the initialization phase and incremental and decremental parts are handled separately.

Degrees of Freedom. One degree of freedom is the choice of the datastructure Q used in the main phase. This approach requires Q to support the same operations as a priority queue but the specification of operation EXTRACTMIN is relaxed: EXTRACTMIN is allowed to output and remove *any* element currently stored within Q.

Narváez et al. propose three different choices. Firstly, a FIFO-queue which leads to an approach similar to the Bellman-Ford Algorithm (BF) [Bel58]. Secondly a heap, either implemented as a binary heap or as a linked list. Thirdly a D'Esopo-Pape-like approach [Pap74] is stated: The datastructure stores the elements in an ordered list. Elements are always extracted from the top. When inserting an element v for the first time into Q it is appended at the bottom, all following times v is appended at the top.

In our experiments we do not include the D'Esopo-Pape variants of the NARVÁEZ-framework because pretests had revealed some instances with extremely bad performance with this approach. For the heap variants we only report the results for the binary heap implementation as this variant was clearly superior.

The other degree of freedom consists of two different variants for the main phase of the algorithms. We describe both variants below and refer to the specific algorithms as NAR{1st, 2nd}{HEAP, BF}.

Overall Workflow. First, the batch update is split into the set of updates with edge weight increases (the decremental part) and the set of updates with edge weight decreases (the incremental part). Then, the initialization phase of the incremental part is executed, followed by the main phase. Afterwards, the initialization phase of the decremental part is executed, again followed by the main phase.

Outline. The algorithm maintains a shortest-paths tree T of the graph by storing the parent node $P[v]$ for each node v.

Initialization - Decremental. The decremental part of the initialization phase first applies each edge weight increment to the graph and updates the labels $D[]$ by, for each node v, setting $D[v]$ to be the distance from s to v in the outdated shortest-paths tree T. Afterwards, for each node y with changed tentative distance, we relax each incoming edge (x, y) and insert y with priority $\mathrm{con}(y)$ into Q if y is inconsistent.

Initialization - Incremental. The incremental part of the initialization phase first applies all edge weight decrements to the graph and updates the labels $D[]$ by, for each node v, setting $D[v]$ to be the distance from s to v in the outdated shortest-paths tree T. Afterwards, for each node x with changed tentative distance, we relax all outgoing edges (x, y) and insert y with priority $D[x] + \mathrm{len}(x, y)$ into Q if (x, y) is underconsistent.

Main Phase. The main phase of the 1st variant basically works like Dijkstra's algorithm. There are two differences: Firstly, the distance label $D[x]$ of a node x is not altered when relaxing edges but altered to the priority $\mathrm{key}(x)$ of node x directly after extracting x from Q. Secondly, the decision which node to process next depends on the type of Q.

The main phase of the 2nd variant uses the following additional heuristic: After extracting node x from the queue, $D[x]$ is updated. This distance change of x is propagated in the subtree of T starting with x. Branches with already better distance label in Q are omitted. Further, nodes v are deleted from Q for which the priority $\mathrm{key}(v)$ of v in Q is bigger than the new value of $D[v]$: For such nodes, the value in the queue does not improve the tentative distance anymore. Afterwards, not only edges outgoing of x, but all edges outgoing of this subtree are relaxed.

The recomputation of T is done by setting the parent of x accordingly when extracting node x from Q and changing $D[x]$. To that end, the new parent is already stored when x is inserted into Q and changed, whenever the priority of x is changed.

Auxiliary Data. The algorithm maintains a shortest-paths tree T of the graph by storing the parent node $P[v]$ for each node v. Initially, T is a shortest-paths tree with respect to the graph before the update, after execution of the algorithm, T is a shortest-paths tree with respect to the graph after the update. With $B(u, v)$ we denote the set of nodes that are contained in the branch of T that starts with (u, v) minus $\{u\}$.

Further, along with any node v in Q a potential new parent parent(v) in the shortest-paths tree is stored. Hence Insert is expanded to Insert$(v, \text{key}(v), \text{parent}(v))$, DecreaseKey to DecreaseKey$(v, \text{key}(v), \text{parent}(v))$, and InsertOrUpdate to InsertOrUpdate$(v, \text{key}(v), \text{parent}(v))$. Finally, extractMin additionally outputs key(v) and parent(v).

Initialization Phase - Decremental Part. For each edge (u, v) with weight increase λ, we update len(u, v) and set $D[x] := D[x] + \lambda$ for each x in $B(u, v)$. By N_{inc} we denote the set of all vertices x which are contained in at least one of the sets $B(u, v)$ considered in the decremental part of the update. Afterwards we relax each edge with target y in N_{inc}. Let y be an underconsistent node in N_{inc} and $(x, y) \in E$ be the incoming edge of y such that $D[x] + \text{len}(x, y)$ is minimal. We call InsertOrUpdate$(y, D[x] + \text{len}(x, y), x)$.

Initialization Phase - Incremental Part. For each edge (u, v) with weight decrease λ, we update len(u, v) and set $D[x] := D[x] - \lambda$ for each x in $B(u, v)$. By N_{dec} we denote the set of all vertices x which are contained in at least one of the sets $B(u, v)$ considered in the incremental part of the update. Afterwards we relax each edge (x, y) with source x in N_{dec}. Let y be an underconsistent node in $V \setminus N_{dec}$ and $(x, y) \in E$ be the incoming edge of y such that $D[x] + \text{len}(x, y)$ is minimal. We call InsertOrUpdate$(y, D[x] + \text{len}(x, y), x)$.

Main Phase, 1st Variant. We perform as follows until Q is empty: We extract and delete the next triple $(v, \text{key}(v), \text{parent}(v))$ from Q (according to the actual choice of Q), set $D[v] := \text{key}(v)$ and $P(v) := \text{parent}(v)$. Afterwards we relax each outgoing edge (v, w) and call InsertOrUpdate$(w, D[v] + \text{len}(v, w), v)$ if (v, w) is underconsistent.

Main Phase, 2nd Variant. We perform as follows until Q is empty: We extract and delete the next triple $(v, \text{key}(v), \text{parent}(v))$ from Q (according to the actual choice of Q). We set $\lambda := \text{key}(v) - D[v]$ and $P(v) := \text{parent}(v)$.

Now, we identify a subset N consisting of v and the set of all descendants of v in T as follows: We perform a depth first search in T starting at v. If we visit a node u that is already contained in Q with priority key$(u) < D[u] + \lambda$ we do not include u in N and do not investigate edges outgoing from u. If we visit a node u that is already contained in Q with priority key$(u) \geq D[u] + \lambda$ we remove u from Q. After termination of the depth first search, N consists of all vertices visited during the search.

Afterwards we set $D[v] := D[v] + \lambda$ for each node $v \in N$. Then, we relax each edge (v, w) outgoing from a node $v \in N$ and call InsertOrUpdate$(w, D[v] + \text{len}(v, w), v)$ if (v, w) is underconsistent.

5.4 Tuning SWSF-FP

In this section, we review the batch algorithm SWSF-FP which is due to Ramalingam and Reps [RR96b] and propose some tuned variants of it. The algorithms described in this section are the only algorithms in our testset that process updates completely in a batch. The main difference to the algorithms described in the last section are the limitations on the values $D[v]$. The initialization phase of SWSF-FP does not assure that $D[v]$ contains upper bounds for the distance from s to v in the updated graph. Our description only considers edge weight increases and decreases. The generalization to edge insertions and deletions is straightforward. We use the terminology as introduced in Section 5.3.1.

5.4.1 SWSF-FP.

Outline. This approach attaches an additional label $d[v]$ to each node v and, after the initialization phase, incorporates two invariants:

- for each $v \in V$, the value $d[v]$ equals the consistent value $\mathrm{con}(v)$ of v
- each inconsistent node v is contained in the priority queue Q.

If we change the graph or the array $D[]$ we may violate one of these invariants for a node v and have to repair them. This is done by the routine adjust. Pseudocode for that routine is given as Algorithm 5.1. The initialization phase does nothing than adjusting all target nodes of updated edges.

A straightforward main phase could work as follows: Iteratively extract an arbitrary node from Q, set $D[v] := d[v]$ and adjust all targets of outgoing edges of v. After termination, $D[]$ contains correct distances as the Bellman-Ford Equations are fulfilled for any node.

A problem of this approach is, that we cannot bound the number of times that a node gets re-inserted into the queue. Hence, the algorithm SWSF-FP applies two changes: Firstly, if we extract a node v with $D[v] < d[v]$, we set $D[v] := \infty$ and adjust v and all of its outgoing neighbors. Secondly, we do not extract an arbitrary node from Q but a node v minimizing the value $\min\{d[v], D[v]\}$. One can show that, this way, each node is re-inserted into Q at most once.

Auxiliary Data. The approach does not require any auxiliary data. In order to initialize the vector $d[]$ that contains the consistent value of each node one could simply copy the distance vector $D[]$. In order to save time for the copy process we implemented $d[]$ as auxiliary data.

Description. We say we *adjust an inconsistent node* v when we set $d[v] := \mathrm{con}(v)$ and insert v with priority $\min(D[v], d[v])$ in Q. In case v is already contained in Q we only update the priority. We *adjust a consistent node* v when we remove it from Q. If v is not contained in Q we do nothing.

Initially, we adjust each node which is target of an edge in U. *Main Phase.* While Q is not empty, we perform as follows: We extract and delete the minimum node w from Q. If $d[w] < D[w]$ we set $D[w] := d[w]$ and adjust each outgoing neighbor of w. If $d[w] > D[w]$ we set $D[w] := \infty$ and adjust w and each of its outgoing neighbors. Pseudocode for that approach is given as Algorithm 5.2.

5.4.2 Tuned SWSF

This algorithm basically works like the SWSF-FP-algorithm, but with less computational effort.

Algorithm 5.1: adjust(v)

```
input  : node $v$
access : to all data of the calling SWSF-FP-routine
1 if $v = s$ then break                          /* break as $D[s]$ always equals 0 */
2 $d[v] \leftarrow \infty$                                    /* compute consistent value of $v$ */
3 for each incoming edge $(u, v)$ do
4     $d[v] \leftarrow \min\{d[v], D[u] + \text{len}(u, v)\}$
5 if $d[v] \neq D[v]$ then                                   /* if $v$ is not consistent */
6     $Q$.InsertOrUpdate($v, \min\{d[v], D[v]\}$)
7 else                                                       /* if $v$ is consistent */
8     if $Q$.contains($v$) then $Q$. remove($v$)
```

Algorithm 5.2: SWSF-FP

```
input  : graph $G = (V, E, len)$
         source $s \in V$
         distance vector $D[]=d[]$ containing distances from $s$ in $G = (V, E, \text{len})$
         set of updated edges $U$
         new length function $len_{new} : E \to \mathbb{R}_{\geq 0}$
output: distance vector $D[]=d[]$ containing distances from $s$ in $G = (V, E, \text{len}_{\text{new}})$
uses   : priority queue $Q$
1 $\text{len} \leftarrow \text{len}_{\text{new}}$                                   /* Initialization Phase */
2 for each node $y$ such that there is an edge $(x, y) \in U$ do
3     adjust($y$)
4 while not $Q$.isEmpty do                                   /* Main Phase */
5     $v \leftarrow Q$.extractMin
6     if $d[v] < D[v]$ then
7         $D[v] \leftarrow d[v]$
8         for each outgoing edge $(v, w) \in E$ do
9             adjust($w$)
10    if $d[v] > D[v]$ then
11        $D[v] \leftarrow \infty$
12        for each outgoing edge $(v, w) \in E$ do
13            adjust($w$)
14        adjust($v$)
```

Outline. We have a look at the algorithm SWSF-FP. In lines 1, 7 and 11, we change either edge weights or values in $D[]$. Hence, in lines 3, 9, 13 and 14 we recompute the consistent values of possibly affected nodes. To that end, we scan all incoming edges of these nodes. TUNED SWSF tries to scan fewer of such incoming edges. We distinguish four cases:

We decrease the weight of an edge (x, y). This operation can only violate the consistency of node y. As decreasing the edge weight may only decrease the consistent value of y, it is $\min\{d[y], D[x] + \text{len}(x, y)\}$ the consistent value of y after the weight change.

We increase the weight of an edge (x, y). Again, this operation can only violate the consistency of node y. Before updating the weight, we check if $D[x] + \text{len}(x, y) > d[y]$. In that case there is another edge (u, y) such that $D[u] + \text{len}(u, y) = d[y]$ and $D[x]$ is not responsible for the consistent value of y. Hence we do not have to recompute $D[y]$.

We decrease $D[x]$. This may affect any target y of an outgoing edge (x, y). The consistent value of y can only decrease and $\min\{d[y], D[x] + \text{len}(x, y)\}$ is the consistent value of y after changing $D[x]$.

We increase $D[x]$. Also for that case, only nodes y that are target of an outgoing edge (x, y) may become inconsistent. Before we update $D[x]$, we check if $D[x]+\text{len}(x, y) > d[y]$. In that case $D[y]$ does not change, as seen for the case of an edge weight increase.

Auxiliary Data. Same as for SWSF-FP.

Description. The approach can be constructed by exchanging any adjust-operation in SWSF-FP by the corresponding argument of the above case distinction. The pseudocode is given as Algorithm 5.4, the subroutine $\text{con}(v)$ as Algorithm 5.3. The correctness follows directly from the correctness of SWSF-FP and the case distinction.

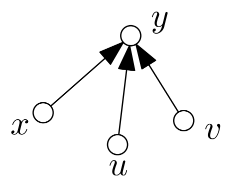
Algorithm 5.3: $\text{con}(v)$

```
input  : node $v$
access : to all data of the calling TUNED SWSF-routine
output: consistent value of node $v$

if $v = s$ then                          /* consistent value of $s$ is 0 */
    return 0
else                                     /* otherwise scan all incoming edges */
    conValue($v$) ← $\infty$
    for each incoming edge $(u, v) \in E$ do
        conValue ← min{conValue, $D[u] + \text{len}(u, v)$}
    return conValue
```

5.4.3 Tuned SWSF-RR

This variant enhances the TUNED SWSF-algorithm with a technique adapted from the RR-algorithm.

Outline. The improvement takes place at line 22 in TUNED SWSF. We have just extracted a node x from the queue and increased the tentative distance from $D_{old}[x]$ to $D[x]$. This may destroy consistency of node y for each outgoing edge (x, y) with $D_{old}[x] + len(x, y) = d[y]$. Hence we have to recompute the consistent value of y. If we knew that there is another node u with $D[u] + len(u, y) = d[y]$ we could skip the



Algorithm 5.4: Tuned SWSF

```
input  : graph $G = (V, E, len)$
         distance vector $D[]=d[]$ containing distances from $s$ in $G = (V, E, \text{len})$
         source $s$
         set of updated edges $U$
         new length function $len_{new} : E \to \mathbb{R}_{\geq 0}$
output: distance vector $D[]=d[]$ containing distances from $s$ in $G = (V, E, \text{len}_{\text{new}})$
uses   : priority queue $Q$

1  for $(u, v) \in U$ do                                   /* Initialization Phase */
2      if $\text{len}(u, v) > \text{len}_{\text{new}}(u, v)$ then
3          $\text{len}(u, v) \leftarrow \text{len}_{\text{new}}(u, v)$
4          $d[v] \leftarrow \min\{d[v], D[u] + \text{len}(u, v)\}$
5      if $\text{len}(u, v) < \text{len}_{new}(u, v)$ then
6          $\text{len}_{\text{old}} \leftarrow \text{len}(u, v)$
7          $\text{len}(u, v) \leftarrow \text{len}_{\text{new}}(u, v)$
8          if $D[u] + \text{len}_{\text{old}} = d[v]$ then $d[v] \leftarrow \text{con}(v)$
9      if $D[v] \neq d[v]$ then
10         $Q$.InsertOrUpdate($v, \min\{D[v], d[v]\}$)
11 while not $Q$.isEmpty do                                /* Main Phase */
12     $v \leftarrow Q$.ExtractMin
13     if $d[v] < D[v]$ then
14         $D[v] \leftarrow d[v]$
15         for each outgoing edge $(v, w) \in E$ do
16             if $D[v] + \text{len}(v, w) < d[w]$ then
17                 $d[w] \leftarrow D[v] + \text{len}(v, w)$
18                 $Q$.InsertOrUpdate($w, \min\{D[w], d[w]\}$)
19     if $d[v] > D[v]$ then
20         $D_{old} \leftarrow D[v]$
21         $D[v] \leftarrow \infty$
22         for each outgoing edge $(v, w) \in E$ with $D_{\text{old}} + \text{len}(v, w) = d[w]$ do
23             $d[w] \leftarrow \text{con}(w)$
24             $Q$.InsertOrUpdate($w, \min\{D[w], d[w]\}$)
25         $d[v] \leftarrow \text{con}(v)$
26         $Q$.InsertOrUpdate($v, \min\{D[v], d[v]\}$)
```

computation of $\mathrm{con}(y)$. To that end we maintain the subgraph $\mathcal{S}$ consisting of all edges (u, v) with $D[u] + \mathit{len}(u, v) = d[v]$ and, for each node w, the indegree of w on $\mathcal{S}$. With this information we can do better: We only recompute $\mathrm{con}(y)$ if the indegree of y in $\mathcal{S}$ becomes zero.

Auxiliary Data. Same as for SWSF-FP and as follows: For each edge (u, v), a boolean label $\mathcal{S}(u, v)$ indicating if $D[u] + \mathrm{len}(u, v) = d[v]$. We denote the subgraph induced by all edges (u, v) with $\mathcal{S}(u, v) = \mathrm{true}$ by $\mathcal{S}$. For each node v, a label $\mathrm{indeg}(v)$ indicating the indegree of v in $\mathcal{S}$. Note, that $\mathcal{S}$ initially equals the outdated shortest-paths subgraph.

Description. The algorithm performs like Tuned SWSF with the following two differences: Line 22 of Tuned SWSF is changed to:

for each outgoing edge $(v, w) \in E$ *with* $D_{\mathrm{old}} + \mathit{len}(v, w) = d[w]$ *and* $\mathrm{indeg}(w) = 0$.

Further we have to keep track how changes of len, $D[]$ and $d[]$ effect in changes of $\mathcal{S}$. The following cases may occur:

- *We increase* $D[v]$ *to* $D_{\mathrm{new}}[v]$. For each outgoing edge $(v, w) \in E$ with $D[v] + \mathrm{len}(v, w) = d[w]$ we remove (v, w) from $\mathcal{S}$ and decrement $\mathrm{indeg}(w)$.
- *We decrease* $D[v]$ *to* $D_{\mathrm{new}}[v]$. For each outgoing edge $(v, w) \in E$ with $D_{\mathrm{new}}[v] + \mathrm{len}(v, w) = d[w]$ we set $\mathrm{indeg}(w) = 1$, remove all incoming edge of w from $\mathcal{S}$ and insert (v, w) in $\mathcal{S}$.
- *We increase* $d[v]$ *to* $d_{\mathrm{new}}[v]$. This happens after completely recomputing $\mathrm{con}(v)$. Hence, while recomputing $\mathrm{con}(v)$ we remove all incoming edges of v from $\mathcal{S}$ and insert the corresponding new edges and the indegree accordingly.
- *We decrease* $d[v]$ *to* $d_{\mathrm{new}}[v]$. Then, we already know the edge (u, v) responsible for the decrease operation. Hence, we remove all incoming edges of v from $\mathcal{S}$, insert (u, v) into $\mathcal{S}$ and set $\mathrm{indeg}(v) = 1$.
- *We increase* $\mathrm{len}(u, v)$ *to* $\mathrm{len}_{\mathrm{new}}(u, v)$. We remove (u, v) from $\mathcal{S}$ and decrement $\mathrm{indeg}(v)$ if $D[u] + \mathrm{len}(u, v) = d[v]$.
- *We decrease* $\mathrm{len}_{\mathrm{new}}(u, v)$. If $D[u] + \mathrm{len}_{\mathrm{new}}(u, v) = d[v]$ we remove all incoming edge of v from $\mathcal{S}$, insert (u, v) in $\mathcal{S}$ and set $\mathrm{indeg}(v) = 1$.

5.4.4 Tuned SWSF-NAR

This variant enhances the Tuned SWSF-algorithm with a technique adapted from the Narváez-algorithm.

Outline. We additionally store a shortest-paths tree of the graph. The initialization phase first recomputes distances in the graph with respect to the *new* edge weights but the *old* shortest-paths tree. Then, all possibly inconsistent nodes are adjusted. The main phase of the algorithm works like the main phase of Tuned SWSF but additionally has to maintain the shortest-paths tree.

Auxiliary data. Same as for SWSF-FP and as follows: For each node v that is not the source, a label $P[v]$ is given pointing at another node, such that $D[P[v]] + \mathrm{len}(P[v], v) = d[v]$. We denote by T the subgraph given by all edges $(P[v], v)$. Initially, T is a shortest-paths tree in the original graph.

Description. The algorithm works like the Tuned SWSF with the following two differences: Firstly, whenever $d[v]$ or $D[w]$ is changed, the value $P[v]$ is changed accordingly.

Secondly, the initialization phase is different and works as follows: We start by updating the edge weights. Then, we recompute distances in the graph with respect to the *new* edge weights but the *old* shortest-paths tree. This is done by starting a depth first search in T for each target node of an updated edge on T. In order to process each node only once, we process the updates ordered by the distance of their source node and do not visit already visited nodes again. When visiting a node v we set $D[v] := D[P[v]] + \text{len}(P[v], v)$.

Finally we adjust each node that either is target node of an updated edge, is marked as visited or is target of an edge whose source node has been visited. The pseudocode of the initialization phase is given as Algorithm 5.5. Note that the FIFO F can easily be implemented to still contain all visited nodes at Line 13.

Algorithm 5.5: Initialization Phase of Tuned SWSF-NAR

```
uses: FIFO F
1  len ← $\text{len}_{\text{old}}$
2  remove all edges with target $s$ from $U$
3  sort $U = \{(u_1, v_1), \ldots, (u_k, v_k)\}$ such that $D[v_i] \leq D[v_{i+1}]$
4  for $i = 1, \ldots, k$ do
5      if $P[v_i] = u_i$ and $v_i$ is not marked as visited then
6          $F$.PUSH_BACK($v_i$)
7      while not $F$.IsEmpty do
8          $v \leftarrow F$.POP_FRONT
9          mark $v$ as visited
10         $D[v] \leftarrow D[P[v]] + \text{len}(P[v], v)$
11         for each outgoing edge $(v, w) \in E$ with $P[w] = v$ do
12             $F$.PUSH_BACK($v$)
13 for each node $v$ that either is visited or target node of an updated edge or target
   node of an edge whose source is visited do
14     adjust($v$)
```

5.5 Experiments

In this section, we present an experimental evaluation of the algorithms described above. Our implementation is written in C++ (using the STL at some points). Our tests were executed on one core of an AMD Opteron 2218, running SUSE Linux 10.3. The machine is clocked at 2.6 GHz, has 32 GB of RAM and 2 x 1 MB of L2 cache. The program was compiled with GCC 4.2, using optimization level 3.

For each experiment, 1000 update instances were generated. Each instance is generated by choosing a new source node uniformly at random and choosing the edge updates according to the given distribution. To properly measure the speedups, a full Dijkstra run is performed directly after each update and the speedup compared to that run (i.e., the time needed by Dijkstra's algorithm divided by the time needed by the update algorithm) is computed. Finally we compute the mean value of these speedups. Thus, measurement disturbances due to background processes, etc are avoided as much as possible. For Tables 5.2-5.6 we showed in bold letters all algorithms whose performance was at least 85% of the best observed performance. We later see that some algorithms perform quite similarly. In this case we only report the results for one example algorithm. The full data is given in the appendix.

In our experiments we evaluated all previously described algorithms. We do not include the heuristic of Buriol et al. [BRT08] because it does not support edge insertions or deletions. To gain further insights into the performance of the batch-algorithms (Narváez and Tuned SWSF), we executed these two times: one time with processing the edges in batch and one time with iteratively processing the edges one after another. We refer to these approaches as itNar and Ittuned SWSF. Note that we refer to the Narváez-framework as a batch algorithm while it actually does not perform updates completely in a batch: its initialization phase handles edge updates iteratively but the following main phase handles all updates in a batch.

5.5.1 Graph Instances

We use the following test instances for our study. The average absolute running times of Dijkstra's algorithm for each instance is given in Table 5.1.

UNIT-DISK. During the last years, the field of sensor networks has received wide attention. We evaluate so called unit-disk graphs, which are widely used for experimental evaluations in that field. Given numbers n and m, a unit-disk graph is generated by randomly assigning each of the n nodes to a point in the unit square of the Euclidean plain. Two nodes are connected by an edge in case their Euclidean distance is below a given radius. This radius is adjusted such that the resulting graph has approximately m edges. As edge weights we use the Euclidean distance to the power of 0 (hop length), 1 (Euclidean distance) and 2 (energy). All tested graphs consist of 15 000 nodes.

RAILWAY. The graph RAIL represents the condensed railway network of Europe, based on timetable information provided by the company HaCon [HaC08] for scientific use. Nodes represent stations while edges represent direct connections between the stations. The edge weights correspond to the average travel time between the according stations. The graph has 29 578 nodes and 159 914 edges.

AS-GRAPH. The graph AS-HOP represents the internet as of 2008/3/26 on the AS-level, i.e., each node corresponds to an autonomous system and edges represent connections between autonomous systems. This graph is taken from the Routeviews project page [Uni08]. It has 27 909 nodes and 114 474 edges. The edge weight is 1 for each edge. The

graph description	name	runtime
road network of Luxembourg	LUX	7.09
road network of the Netherlands	NLD	429.99
road network of Germany	DEU	2414.28
condensed railway network of Europe	RAIL	7.86
(part of the) internet on router-level	CAIDA	157.85
internet on AS-level, edge length 1	AS-HOP	8.80
internet on AS-level, random edge lengths	AS-RAN	17.53
synthetic grid-graph, 10.000 nodes	GRID 100	2.03
synthetic grid-graph, 90.000 nodes	GRID 300	26.86
UNIT-DISK graph, hop length, av. edge degree 7	UN-HO	5.39
UNIT-DISK graph, hop length, av. edge degree 10	-	6.61
UNIT-DISK graph, hop length, av. edge degree 15	-	8.91
UNIT-DISK graph, Euclidean length, av. edge degree 7	UN-EU	5.84
UNIT-DISK graph, Euclidean length, av. edge degree 10	-	7.24
UNIT-DISK graph, Euclidean length, av. edge degree 15	-	9.34
UNIT-DISK graph, energy length, av. edge degree 7	-	6.12
UNIT-DISK graph, energy length, av. edge degree 10	-	7.85
UNIT-DISK graph, energy length, av. edge degree 15	-	10.35

Table 5.1: Average absolute runtime (in milliseconds) of a full run of Dijkstra's algorithm

same graph with edge weights chosen uniformly at random from the interval $[1, 1000]$ is called AS-RAN.

CAIDA. This dataset represents the internet on router level, i.e., nodes are routers and edges represent connections between routers. The network is taken from the CAIDA webpage [CAI08] and has 190 914 nodes and 1 215 220 edges. The edge weight is 1 for each edge.

ROAD. We evaluate three road networks provided by the PTV AG [PTV08]. DEU represents Germany with 4 378 447 nodes and 10 968 884 edges, NLD the Netherlands with 946 632 nodes and 2 358 226 edges and LUX represents Luxembourg with 30 647 nodes and 75 576 edges. The edge weights are the corresponding travel times with speed profile 'slow car'.

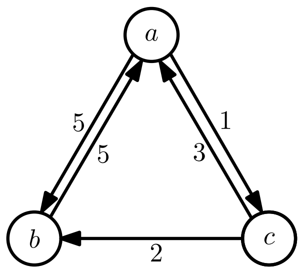
GRID. These are fully synthetic graphs based on two-dimensional square grids. The nodes of the graph correspond to the crossings in the grid. There is an edge between two nodes if these are neighbors on the grid. Edge weights are randomly chosen integer values between 1 and 1000. GRID 100 is a 100x100 grid graph while GRID 300 is a 300x300 grid graph.

5.5.2 Data Structures

For our tests we always apply integral edge weights. We further use a binary heap whenever a priority queue is needed. We use the graph data structure that is also applied in [BD08, BDW07, Del08]. There, the data structure has experimentally shown to perform well in the context of shortest-paths computation on sparse graphs. It is described in the following.

The input graph $G = (V, E, \text{len})$ is stored by mixing forward and reverse representation, an example is given as Figure 5.1. It is represented by two arrays. Nodes are directly identified by the numbers $0, \dots, |V| - 1$. Attributes of nodes like distances are stored in arrays and are keyed by the number of the corresponding node.

We use two arrays to manage the edges. The array EDGES contains the actual infor-



NODES	
array index represents	edges index
0 *a*	0
1 *b*	3
2 *c*	5

EDGES					
edge index	target node	edge length	reverse index	forward	backward
0	1	5	3	✓	✓
1	2	1	5	✓	–
2	2	3	6	–	✓
3	0	5	0	✓	✓
4	2	2	7	–	✓
5	0	1	1	–	✓
6	0	3	2	✓	–
7	1	2	4	✓	–

Figure 5.1: Example graph and the corresponding data structure. Information given in the grey areas is not part of the data structure but added for better readability.

mation on the edges like length or target node. Each edge (u, v) corresponds to two entries in EDGES: one as a forward edge of node u (the *forward entry*) and one as a backward edge of node v (the *backward entry*). The entries of (u, v) are of the following form:

- forward entry:
  $(v$, len(u, v), false, true, index of the backward entry of (u, v) in EDGES$)$
- backward entry:
  $(u$, len(u, v), true, false, index of the forward entry of (u, v) in EDGES$)$

The third (fourth) component of an entry indicates if the entry is a backward (forward) entry.

The array EDGES is ordered such that, for any node v, the forward entries of all outgoing edges of v and the backward entries of all incoming edges of v are located in a row. We call such a row the *block of* v. Moreover, these blocks are ordered such that the block of a node x is located before the block of a node y if $x < y$ (remember that we identify nodes directly by their numbers).

To be able to iterate efficiently over the adjacent edges of a given node, we use the second array NODES. The array contains, for each node v, the index NODES(v) of the first entry of the block of v. In order to iterate over all in- and outcoming edges of node v, one has to consider all entries in EDGES at the locations from NODES(v) to NODES$(v+1)$-1. To recognize the orientation one has to additionally evaluate the 3rd and 4th component of the respective entry. A dummy node is added at the end to simplify the handling of the last node.

Note that source and target node of an edge are not stored in the same entry. Given the forward entry of an edge (u, v), the target node v is directly given as first component of that entry. In order to reconstruct the source node u, one hast to first access the corresponding backward entry. This can be done by using the 5th component of the forward entry.

We often work on graphs that are 'almost undirected'., i.e., on directed graphs for which an edge (u, v) is usually complemented by an edge (v, u) of same length. To save space (and time needed for iterating), we compress the respective entries. More detailed, for edges (u, v) and (v, u) with len$(u, v) =$ len(v, u) we combine the entries

- forward entry of (u, v): $e_1 = (v$, len(u, v), false, true, index of $e_2)$
- backward entry of (u, v): $e_2 = (u$, len(u, v), true, false, index of $e_1)$

- forward entry of (v, u): $e_3 = (u,\ \text{len}(u, v),\ \text{false},\ \text{true},\ \text{index of } e_4)$
- backward entry of (v, u): $e_4 = (v,\ \text{len}(u, v),\ \text{true},\ \text{false},\ \text{index of } e_3)$

to

- combined forward entry of (u, v) and backward entry of (v, u):
  $e_5 = (v, \text{len}(u, v),\ \text{true},\ \text{true},\ \text{index of } e_6)$
- combined forward entry of (v, u) and backward entry of (u, v):
  $e_6 = (u,\ \text{len}(u, v),\ \text{true},\ \text{true},\ \text{index of } e_5)$.

In order to dynamically insert or delete edges one has to rearrange the array EDGES and adjust the information in NODES. To prevent that as far as possible some dummy edges are inserted in EDGES and some extra information is maintained to organize the dummy edges. We do not describe that in full detail as this has to be equally done for each update algorithm and for a full recomputation from scratch.

5.5.3 Assessing the Performance of the Algorithms

Let $U = \{u_1, \ldots, u_k\}$ be a set of edge updates. By $\Delta(G, U)$ we denote the number of vertices in V for which the distance from the source changes due to the batch update U. The *expected speedup* of an update is the number of vertices in the graph divided by $\Delta(G, U)$. This value is roughly the speedup we expect from a good update algorithm. Of course, speedups can even be higher for special instances. It experimentally turned out that the propagation of the updated edge's weights through the original shortest-paths tree can gain a large speedup especially when the topology of the original shortest-paths tree does not change.

When we want to measure the difficulty of an update for an iterative algorithm we consider $U = (u_1, \ldots, u_k)$ to be ordered. We perform the updates u_i iteratively in the given order (always additional to the former updates) obtaining a sequence of graphs $G = G_0, G_1, \ldots, G_k$. We write $\delta(G, (u_1, \ldots, u_k)) := \sum_{i=0}^{k-1} \Delta(G_i, \{u_{i+1}\})$. We have the following hypothesis: if the difference between $\Delta(G, U)$ and $\delta(G, U)$ is small, then the contained single-edge updates do not interfere much and it is reasonable to use an iterative algorithm for the update. If the difference is great, an iterative algorithm would change the distance of many nodes multiple times. Hence, it is more appropriate to use a batch algorithm. The experimental evaluation will support our hypothesis.

The algorithm ITTUNED SWSF can be seen as a very simple iterative approach incorporating no extra features like early edge-weight propagation. Figure 5.2 shows the improvement of TUNED SWSF over ITTUNED SWSF compared with $\Delta(G, U)/\delta(G, U)$ for all batch experiments performed in this study.

5.5.4 Space-Saving Implementation of RR

The algorithm RR needs to maintain the shortest-paths subgraph $\mathcal{S}$. This subgraph is implicitly given by each edge (u, v) with $D[u] + len(u, v) = D[v]$. We implemented the algorithm doubly. One time we explicitely stored the subgraph (RR DAG), i.e., we stored a flag for each edge (u, v) indicating if $(u, v) \in \mathcal{S}$. The other time with reconstructed it when needed (RR), i.e., we checked if $D[u]+len(u, v) = D[v]$ in order to know if $(u, v) \in \mathcal{S}$. It turned out that there are only small differences between both implementations, with no variant being clearly superior. We therefore only report the results for the space-saving implementation RR.

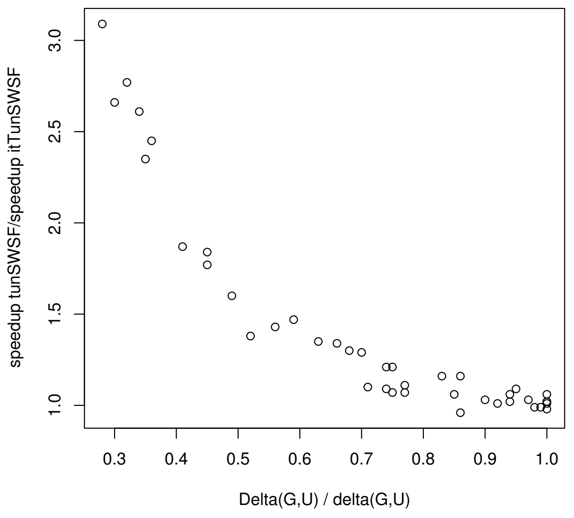


Figure 5.2: Value of $\Delta(G, U)/\delta(G, U)$ (x-axis) and speedup of Tuned SWSF / speedup of It-tuned SWSF (y-axis) for all batch-experiments.

	LUX	NLD	DEU	RAIL	CAIDA	AS-HOP	AS-RAN	GR100	GR300	UN-HO	UN-EU
FMN	42	1504	29087	151	22702	1624	2182	25	142	327	36
SWSF-FP	112	3759	65404	366	12429	416	691	59	351	1613	31
tunSWSF-FP	152	**5140**	84873	**562**	16406	893	**3442**	105	598	**2436**	**186**
tunSWSF-NAR	147	3354	70245	215	9306	614	695	94	523	748	129
tunSWSF-RR	118	3798	66068	412	26093	2148	**3766**	74	430	2096	102
RR	155	**4666**	74857	**510**	**34586**	**2599**	**4057**	103	568	**2519**	137
Nar-1st BF	**284**	**5335**	**100944**	357	6578	417	305	**138**	**784**	1176	20
$\Delta(G, U)$	130	141	71	31	0.21	0.41	0.74	59	113	0.01	93
exp. speedup	236	6762	62549	986	inf	inf	inf	169	804	inf	163

Table 5.2: Speedups of experiments with single-edge updates.

5.5.5 Single-Edge Update Experiments

We start our experimental study with single edge updates. Because of a different focus of this work we do not carry out a separate analysis for the decremental and the incremental case. An update consists of choosing an edge uniformly at random and multiplying its weight by a random value in $(0, 2)$. The results can be seen in Table 5.2 , extended results in Table A.1.

We observe that the algorithms of the Narváez-framework have only tiny differences in performance with Nar-1st BF being slightly (but not significantly) faster most times (see Table A.1). There is no such uniform behavior for the SWSF-FP-like algorithms. Tuned SWSF is always faster (between 1.3 and 6 times) than SWSF-FP. The algorithm Tuned SWSF-RR is always at least as fast as SWSF-FP and up to 5.5 times faster. The algorithm Tuned SWSF-NAR seems to be very volatile being between half as fast and 4 times faster than SWSF-FP.

Comparing the different classes of algorithms, we find the algorithms to perform quite differently, but within the same order of magnitude. The algorithm FMN is most times much slower than the other ones. This is due to the overhead caused by maintaining and reading the priority queues used by this algorithm. The technique used in this algorithm can pay off in case nodes with high degree exist (for which many edge-relaxations can be saved). This is not the case for the test instances used. Exceptions are the INTERNET instances CAIDA, AS-HOP and AS-RAN. Here, the gap to the other algorithms is much smaller, which meets the theoretical considerations. Hence, it is to be expected that there are dense graph classes for which FMN is the superior algorithm. On the ROAD and GRID instances, the NARVÁEZ-framework is superior. This is because the structure of the shortest-paths tree stored by the algorithm hardly changes on these experiments. Therefore, the early-propagation of the weight change works well. On the INTERNET instances, RR is the fastest algorithm. Looking at the small value of $\Delta(G, U)$, we can see that updates hardly have any impact on these instances, which favors the RR-algorithm with its small computational overhead and the early detection of edge weight increases that do not change distances in the graph.

The achieved speedups vary greatly between the instances. This is mainly due to the different structure of the underlying graphs, which results in greatly differing expected speedups. It is interesting to see that in nearly all cases the best actual speedups are close to the expected speedups or even higher. This, in combination with the small absolute runtimes, makes us expect that there is not much space for further improvement for the single-edge update case.

The corresponding experiments for edge insertions and deletions are given in Table 5.3, extended results in Table A.2. For this type of experiments the situation is much clearer. The early-detection of updated edges that do not change distances in the graph makes the algorithm RR superior on this testset.

	LUX	NLD	DEU	RAIL	CAIDA	AS-HOP	AS-RAN	GRID100	GRID300	UN-HOP	UN-EU
FMN	62	1884	690	339	21427	1444	3592	50	304	257	681
SWSF-FP	142	4855	1490	742	12218	347	989	99	737	439	543
tunSWSF-FP	190	**6202**	1986	**1049**	15806	767	2151	165	**1185**	873	**1676**
tunSWSF-NAR	154	3128	2326	369	13137	462	769	104	618	428	525
tunSWSF-RR	143	5066	1495	763	27671	1798	4201	120	881	1043	1147
RR	**322**	**6172**	**4619**	**1139**	**34537**	**2427**	**5046**	**221**	**1336**	**1457**	**1714**
Nar-1st BF	55	2504	246	383	9770	280	800	23	105	587	560
$\Delta(G, U)$	52	59	977	7	0.23	0.22	0.12	19	35	2	2
exp. speedup	589	16045	4486	4930	inf	inf	inf	556	2647	7500	15000

Table 5.3: Speedups of single edge updates, edge failure and recovery.

5.5.6 Experiments on Batch Updates

Multiple Randomly Chosen Edges. In this experiment we choose 25 edges uniformly at random. For each edge, we choose a value from the interval $(0, 2)$ uniformly at random and multiply the weight of the edge with that value. The results can be seen in Table A.3. For each graph there is hardly any difference between $\Delta(G, U)$ and $\delta(G, U)$. Therefore, the single-edge updates do only interfere marginally with each other. Hence, not much news is to be expected by this setting regarding the comparison of the algorithms. This has been confirmed by the experiments.

However, we ran the batch-algorithms (NARVÁEZ and TUNED SWSF) twice. One time with processing the edges in batch as stated in the description and one time with iteratively processing the edges one after another. Nearly no runtime differences were observed between the iterative and the batch variants, which indicates a low overhead with batch updates.

	AS-HOP			AS-RAN			CAIDA		
degree	1-10	10-100	100-500	1-10	10-100	100-500	1-10	10-100	100-500
FMN	784	173	23	1368	129	3	7824	2284	185
ittun SWSF-FP	912	228	26	1320	235	11	**12874**	4212	382
SWSF-FP	273	68	8	389	28	1	9651	2203	128
tunSWSF-FP	967	250	24	1417	252	15	**14042**	4693	405
tunSWSF-NAR	407	92	9	410	50	4	9785	2187	122
tunSWSF-RR	**1272**	**528**	**130**	**2475**	**433**	**21**	**12395**	**6839**	**969**
RR	**1438**	**576**	**142**	**2623**	**490**	**17**	**13915**	**7075**	**1163**
Nar-1st Heap	53	21	9	86	59	16	4315	761	75
itNar-1st Heap	52	18	6	71	30	8	4060	573	63
$\delta(G,U)$	1.26	12.16	82.54	1.47	45.01	1365.85	1.97	7.4	90.99
$\Delta(G,U)$	1.07	8.59	71.28	1.1	34.6	712.3	1.45	5.7	85.73
exp. speedup	27909	3489	393	27909	821	86	190914	38183	2246

Table 5.4: Speedups of experiments with node failure and recovery updates on INTERNET-instances.

Node Failure and Recovery. This update class uses the two parameters deg_{min} and deg_{max}. First, a node v with degree between deg_{min} and deg_{max} is chosen uniformly at random. The update consists of two steps. In the first step, v *fails*, i.e., the weights of all edges adjacent to v are set to infinity. In the second step, v *recovers*, i.e., the weights of all edges adjacent to v are reset to their original values. The results can be found in Tables 5.4 and 5.5, extended results in Tables A.4 and A.5. For the sake of completeness we applied this update class also to railway and road networks, the results are given in Table A.7.

We take a look at the INTERNET instances. The most remarkable result is the bad performance of the NARVÁEZ-framework, which clearly is the inferior algorithm for that testset. One main reason for that is, that on this testset the edge-weight propagation in the initialization phase creates useless extra effort which gets overwritten later on. The gap between $\delta(G,U)$ and $\Delta(G,U)$ is small to mid-size, favoring RR with its small overhead, but big enough such that TUNED SWSF-RR is nearly as fast. The small difference of $\delta(G,U)$ and $\Delta(G,U)$ also manifests in the small difference between ITTUNED SWSF and TUNED SWSF.

The situation is similar, but a bit clearer, for UNIT DISK graphs. When applying hop distance, $\delta(G,U)$ and $\Delta(G,U)$ are still near to each other, TUNED SWSF and RR are the best-performing algorithms (with RR being slightly better). When applying Euclidean or energy edge weights, the difference between $\delta(G,U)$ and $\Delta(G,U)$ is much bigger, and TUNED SWSF clearly is the superior algorithm. We also observe the advantage of TUNED SWSF against SWSF-FP being between 2 and 15 times faster.

Traffic Jams. This update class models real-world traffic jams. It derives from the observation that traffic jams often occur along shortest paths. The number k of updated edges is given as a parameter. Initially, a node v is chosen uniformly at random. Then a shortest path P ending at v and containing exactly k edges is chosen uniformly at random. The update consists of two steps: in the first step, the weights of edges in P are multiplied by 10. In the second step, the edge weights are reset to their original values. The results can be found in Tables 5.6 and A.6.

We observe that this update class consists of strongly interfering single-edge updates: there is a big difference between $\delta(G,U)$ and $\Delta(G,U)$. TUNED SWSF and TUNED SWSF-NAR are the best-performing algorithms for this testset. This is because pure batch algorithms avoid processing nodes many times. It is astonishing to see that the NARVÁEZ-

metric	hop			euclidean			energy		
average degree	7	10	15	7	10	15	7	10	15
FMN	30	40	55	27	21	24	12	14	20
ittun SWSF-FP	238	398	485	116	95	98	56	66	91
SWSF-FP	128	214	236	60	32	36	28	24	22
tun SWSF-FP	**260**	**462**	**561**	**158**	**115**	**141**	**75**	**86**	**110**
tun SWSF-NAR	106	116	147	101	77	97	57	61	67
tun SWSF-RR	223	395	527	105	75	89	49	54	67
RR	**289**	**504**	**628**	111	91	106	55	63	84
Nar-1st Heap	70	87	131	84	62	111	52	62	74
itNar-1st Heap	55	71	100	64	50	66	36	46	52
$\delta(G,U)$	19	8	6	86	107	99	194	174	132
$\Delta(G,U)$	18	7	5	54	79	55	128	119	98
exp. speedup	833	2500	3750	283	190	273	117	126	153

Table 5.5: Speedups of experiments with node failure and recovery updates on UNIT DISK-instances.

	GRID			LUX			NLD			DEU		
# edge updates	10	20	30	5	10	20	10	20	30	10	20	30
FMN	3	2	1	4	2	1	11	5	2	185	30	7
ittun SWSF-FP	15	9	5	15	7	2	39	17	6	755	100	23
SWSF-FP	13	10	6	15	9	5	75	32	12	873	173	40
tun SWSF-FP	**23**	**16**	**9**	20	**12**	6	**107**	**44**	**17**	**1210**	235	55
tun SWSF-NAR	**22**	**16**	**9**	**22**	**13**	**7**	**107**	**41**	**17**	**1402**	**342**	**79**
tun SWSF-RR	16	12	7	15	9	5	72	31	12	957	181	42
RR	17	10	5	20	9	3	43	18	6	924	149	36
Nar-1st Heap	16	9	5	20	10	4	37	15	5	1120	196	35
itNar-1st Heap	19	12	6	**24**	**12**	4	57	24	8	**1231**	219	54
$\delta(G,U)$	4367	7909	15552	1178	3052	8616	12910	32088	93725	7885	39260	142191
$\Delta(G,U)$	2591	3564	6412	821	1366	2567	4134	10899	26153	3884	13701	51252
exp. speedup	35	25	14	37	22	12	229	87	36	1127	320	85

Table 5.6: Speedups of experiments with traffic jam updates.

framework is not able to take advantage of the batch-character of the update. This can be seen through a comparison with itNarvaez. The iterative variant is even faster than the batch one, which could be a hint on space for improvement. Again, FMN is much slower than the other algorithms as its overhead does not pay off on these instances.

5.6 Conclusion

In this chapter we focused on the dynamic single-source shortest-paths problem with positive edge weights.

Broad Experimental Study for Single-Edge Case. We gave the first experimental study evaluating the performance for single-edge updates that contains all according algorithms and incorporates a broad set of instance classes. It turned out that the algorithms perform quite differently, but within the same order of magnitude. For road networks and grid graphs, the Narváez-framework performed best while RR was superior for the internet-instances. The Tuned SWSF-algorithm was up to 6 times faster than its base algorithm SWSF-FP. Together with RR it was the best approach for the railway graph and unit-disk graphs. Due to its overhead on the graphs used, FMN was the slowest algorithm. The algorithm RR was superior on all instances when considering edge insertions and deletions instead of edge weight changes.

First Experimental Study for Batch Case. We presented the first experimental study at all for the case of multiple edge-weight changes at a time. One experiment was to choose a set of edges uniformly at random. It turned out that this way the single-edge updates hardly interfere. Therefore, the results deviated not much from the single-edge case. Interestingly, nearly no runtime differences could be observed between the iterative and the batch variants of Tuned SWSF and the Narváez-framework, indicating a low overhead for batch updates.

We also tested two more realistic types of batch updates. One is the simulation of node failure and recovery, which affects all incident edges. When applying that update class, the single-edge updates interfered, but not very strongly. For the internet-instances, the best performing algorithms were RR and Tuned SWSF-RR with RR being slightly faster. For unit disk graphs, Tuned SWSF was the best algorithm with RR being slightly faster when applying uniform edge lengths. The other update class modelled traffic jams. The single-edge updates interfered greatly, Tuned SWSF and Tuned SWSF-NAR were the superior algorithms there.

Tuned Variants of SWSF-FP. We proposed three tuned variants for the SWSF-FP-algorithm and evaluated their performance. Tuned SWSF requires only simple changes compared to SWSF-FP but yields a great improvement in runtime. This approach was never slower than its base algorithm and up to 15 times faster. The algorithm performed very well (and often best) when many nodes were affected by multiple single-edge updates. The combination of Tuned SWSF and ideas of the Narváez-framework was slightly faster than Tuned SWSF for traffic jams, but slower on nearly all other datasets. The enhancement of Tuned SWSF with the main ingredient of the algorithm RR was faster than Tuned SWSF on the INTERNET-instances and slower otherwise.

Further Insights. We introduced a simple methodology (based only on Dijkstra's algorithm) to decide if one should try an iterative or a batch update algorithm for a given instance class. We compared the 'impact' of the update when processed in batch to the 'impact' when processed iteratively. For updates with a big gap between both values, the algorithms Tuned SWSF or Tuned SWSF-RR usually performed best. With a small gap, there was usually a better-performing iterative algorithm.

The achieved speedups varied greatly between different instances. We were able to explain this by measuring the impact of the updates on the graphs. These measurements also proposed that there is not much space for further improvements when applying the instances used in our study.

Summary. We gave an experimental overview on the different approaches for the problem, which can be used as a basis for further research. The most important insight that can be gained from our experiments is the astonishing level of data dependency within the problem. It turned out that a proper assessment of an algorithm's runtime is not possible without full knowledge of the application it is used in. Further, a great amount of experiments is required to get the big picture of an algorithm's efficiency.

Chapter 6

Practical Online Algorithms for Delay Management

The *delay management problem* asks how to react to exogenous delays in public railway traffic such that the overall passenger delay is minimized. These *source delays* occur in the operational business of public transit and easily make the scheduled timetable infeasible. The delay management problem is further complicated by its online nature. Source delays are not known in advance, hence decisions have to be taken without exactly knowing the future. This chapter focuses on *online* delay management.

We enhance established offline models and gain a generic model that is able to cover complex realistic memoryless delay scenarios as well as standard academic delay scenarios that require knowing the past. We introduce and experimentally evaluate online strategies for delay management that are practical, easily applicable, and robust. The most promising approach is based on simulation and a learning strategy. Finally, by analyzing the solutions found, we gain interesting new insights in the structure of good delay management strategies for real-world railway data.

6.1 Introduction

The *delay management problem* asks how to react to exogenous delays (*source delays*) in public railway traffic such that the overall passenger delay is minimized. These source delays occur in the operational business of public transit and usually make the scheduled timetable infeasible. Many operational constraints have to be taken into account when updating the scheduled timetable to a *disposition timetable.* The two main aspects treated in literature are as follows: Firstly, passenger trips often require changing from one train to another. Given a delayed feeding train, a *wait-depart decision* settles the question if a follow-up train should wait in order to enable *changing activities.* Secondly, the limited capacity of the track system complicates the creation of a good disposition timetable. *Headway constraints* model this limited capacity. Every time two trains simultaneously compete for the same part at the train system, it has to be decided which train may go first.

What has been neglected in most publications so far is the online nature of the problem. Source delays are usually not known in advance, hence decisions have to be taken without exactly knowing the future.

Contribution. This chapter focuses on *online* delay management. We enhance both offline models given in [SS10] for the online case: The uncapacitated model that concentrates on wait-depart decisions and the more general model that additionally considers the limited capacity of the track system. We gain a generic model that is able to cover complex realistic memoryless delay scenarios as well as standard academic delay scenarios that require knowing the past.

In particular, we introduce and experimentally evaluate online strategies for delay management. Besides the quality of the online solution, our aim also is to find strategies that are practical in the sense that they are easily applicable and robust. We propose for the first time a learning-strategy for online delay management. It does not need complete information on the state of the entire system and is hence simple and robust, but nevertheless turns out to be superior to other heuristics proposed in the literature and even to ILP-approaches. We compare our results to tight a-posteriori bounds given by an optimal offline solution. Finally, by analyzing the solutions found, we gain interesting new insights in the structure of good delay-management strategies for real-world railway data.

Related work. There exist various models and solution approaches for delay management, mainly treating its offline version. If capacities are neglected the question is to decide which trains should wait for delayed feeder trains and which trains better depart on time. A first integer programming formulation for this problem has been given in [Sch01] and has been further developed in [HGL08] and [Sch07]; see also [Sch06b] for an overview on various models. The complexity of the problem has been investigated in [GGP$^+$04, GJPS05] where it turns out that the problem is NP-hard even in very special cases.

Further publications about delay management include a model in the context of max-plus-algebra [VSM98, Gov98], a formulation as discrete time-cost tradeoff problem [GS07] and simulation approaches [KS10, SMB01]. Recently, also the limited capacity of the track system has been taken into account, see [Sch09a] for modeling issues and [SS10, Sch10] for an extensive analysis of the resulting integer program and heuristic approaches solving the capacitated delay management problem. A model which includes the routing of the passengers can be found in [DHSS09].

An online version of the problem has been studied in [Gat07, GJPW07], where its relation to job-shop scheduling is pointed out and Graham's algorithm is studied. In [BHLS07], it was shown that the online version of the delay management problem is PSPACE-hard. In [KS10], an online version of delay management was studied in which priority-based strategies were compared with the according optimal offline solution and with a solution resulting from an optimal recomputation in each step. The optimal offline solution and the optimal recomputation are gained by an ILP-formulation. The underlying model is similar to the one used in this work. However, it differs in some aspects, such as the objective function and the existence of headway constraints.

Overview. In Section 2 we formally state the problem. Section 3 introduces the considered delay-management strategies. Experimental results are given in Section 4. The chapter ends with a conclusion in Section 5.

6.2 Problem Statement

Given two vectors $a, b \in \mathbb{R}^k$ for some k we write $a \leq b$ if $a_i \leq b_i$ for each $i = 1, \dots, k$. We use the convention $\infty + \infty = \infty$.

Model of the railway system. An *event-activity* network is a directed graph $\mathcal{N} = (\mathcal{E}, \mathcal{A})$ where $\mathcal{E} = \mathcal{E}_{\text{arr}} \dot{\cup} \mathcal{E}_{\text{dep}}$ is decomposed into arrival and departure events $\mathcal{E}_{\text{arr}}$ and $\mathcal{E}_{\text{dep}}$. Each node in $\mathcal{E}$ corresponds to a specific train arriving or departing at a specific time in or from a specific station. An edge in $\mathcal{A}$ is called an *activity.* The set of activities partitions into $\mathcal{A} = \mathcal{A}_{\text{drive}} \dot{\cup} \mathcal{A}_{\text{wait}} \dot{\cup} \mathcal{A}_{\text{change}} \dot{\cup} \mathcal{A}_{\text{head}}$ with each activity $a \in \mathcal{A}$ having a minimal duration $L_a > 0$. Structure and meaning of the activities is as follows:

- A *driving activity* $a \in \mathcal{A}_{\text{drive}} \subseteq \mathcal{E}_{\text{dep}} \times \mathcal{E}_{\text{arr}}$ represents a train driving between two consecutive stations.

- A *waiting activity* $a \in \mathcal{A}_{\text{wait}} \subseteq \mathcal{E}_{\text{arr}} \times \mathcal{E}_{\text{dep}}$ corresponds to the time period in which a train is waiting in a station to let passengers on or off.

- A *changing activity* $a \in \mathcal{A}_{\text{change}} \subseteq \mathcal{E}_{\text{arr}} \times \mathcal{E}_{\text{dep}}$ corresponds to the transfer of passengers from one train to another (by foot, within a station).

- A *headway activity* $a \in \mathcal{A}_{\text{head}} \subseteq \mathcal{E}_{\text{dep}} \times \mathcal{E}_{\text{dep}}$ models the limited capacity of the track system. This can either be two trains driving on the same track into the same direction or two trains driving into opposite directions on a single-way track. The duration $L_{(i,j)}$ of a headway activity (i, j) means that the departure j must take place at least $L_{(i,j)}$ minutes after the departure i (if j actually takes place after i).

For each changing activity $\mathcal{A}_{\text{change}}$ the delay management problem asks to decide if we leave it in the network (i.e., if the connection is maintained) or if it is deleted (if the connection is not maintained). For each activity $(i, j) \in \mathcal{A}_{\text{head}}$, it is $(j, i) \in \mathcal{A}_{\text{head}}$. If (i, j) is respected this means that i happens before j. The delay management problem decides which of these two constraints is respected and which is dropped. There is no such decision to be made for $\mathcal{A}_{\text{drive}}$ and $\mathcal{A}_{\text{wait}}$. Hence we abbreviate $\mathcal{A}_{\text{drive}} \cup \mathcal{A}_{\text{wait}}$ by $\mathcal{A}_{\text{train}}$.

Definition (timetable). A *timetable* for $\mathcal{N}$ is a vector $x \in \mathbb{N}^{|\mathcal{E}|}$ which assigns a time $x_i \in \mathbb{N}$ to each event $i \in \mathcal{E}$ such that

$$x_j - x_i \geq L_{(i,j)} \qquad \text{for each activity } (i,j) \in \mathcal{A}_{\text{train}} \tag{6.1}$$

$$x_j - x_i \geq L_{(i,j)} \text{ or } x_i - x_j \geq L_{(j,i)} \qquad \text{for each activity } (i,j) \in \mathcal{A}_{\text{head}} \tag{6.2}$$

holds.

Throughout this chapter we assume that an event-activity network $\mathcal{N} = (\mathcal{E}, \mathcal{A})$ is given. The network will always be annotated with the following information: For each event $i \in \mathcal{E}_{\text{arr}}$ the number of passengers w_i getting off at event i and arriving there at their final destination, and for all $a \in \mathcal{A}_{\text{change}}$, the number of passengers w_a who want to use activity a. We always assume $w_a > 0$ for each $a \in \mathcal{A}_{\text{change}}$.

Further, a number $T \in \mathbb{N}$ is given. We interpret T as a common time-period for all lines of the underlying railway system und use it as a penalty for not maintaining a changing activity. Finally, we are given a timetable π for $\mathcal{N}$ which we call the *scheduled timetable.* This timetable maintains every changing activity of $\mathcal{N}$, that is, $\pi_i + L_{(i,j)} \leq \pi_j$ for all $(i, j) \in \mathcal{A}_{\text{change}}$. Note that the graph $(\mathcal{E}, \mathcal{A} \setminus \mathcal{A}_{\text{head}})$ is acyclic as otherwise π would not exist. More detailed explanations on this model can be found in [Sch10] and [SS10], a small example of an event-activity network is given in Figure 6.1.

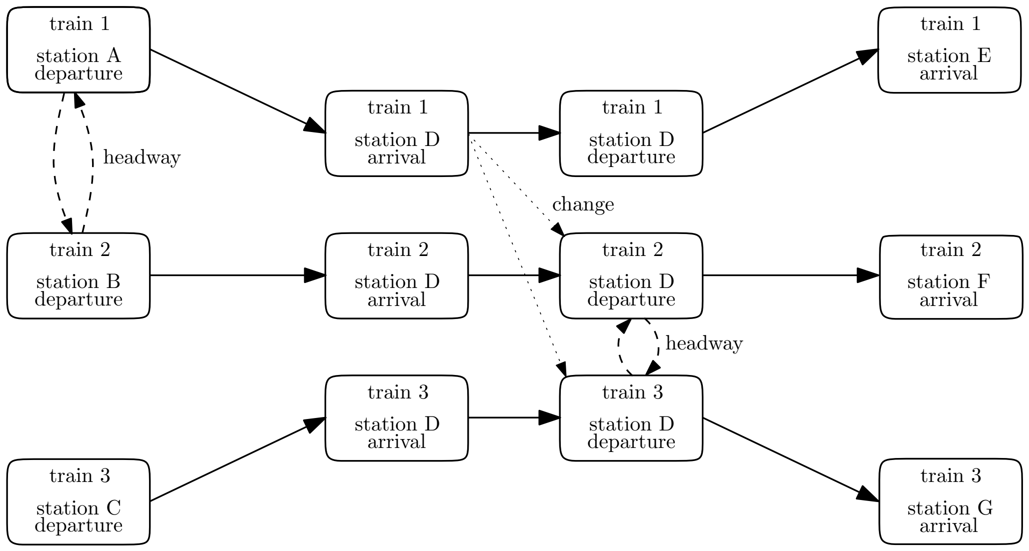


Figure 6.1: Example of an event-activity network. Solid edges represent driving and waiting activities, dotted edges represent changing activities and dashed edges represent headway activities.

Models for offline delay management. A *(source) delay state* d reflects the current knowledge and expectations on exogenous delays in the underlying railway system. Such delays are called *source delays.* Formally, d is a mapping from $\mathcal{E} \cup \mathcal{A}_{\text{train}}$ to $\mathbb{N}$ meaning that there is a source delay of $d(w)$ at activity/event w. We write $d_{(i,j)}$ and d_i for $d(i,j)$ and $d(i)$, respectively. Whenever we become aware of altered source delays, the scheduled timetable π has to be updated to a so-called *disposition timetable* $x \in \mathbb{N}^{|\mathcal{E}|}$.

Definition (Disposition timetable). Given a delay state d, a disposition timetable $x \in \mathbb{N}^{|\mathcal{E}|}$ for d is a timetable for $\mathcal{N}$ such that

$$x_i \geq \pi_i + d_i \qquad \text{for each event } i \text{ in } \mathcal{E} \tag{6.3}$$

$$x_j \geq x_i + d_{(i,j)} + L_{(i,j)} \qquad \text{for each activity } (i,j) \text{ in } \mathcal{A}_{\text{train}}. \tag{6.4}$$

Given a disposition timetable x and a changing activity (i,j) we write

$$z^x_{(i,j)} = \begin{cases} 0 & \text{if } x_j \geq x_i + L_{(i,j)} \\ 1 & \text{otherwise .} \end{cases}$$

That means $z^x_{(i,j)}$ equals 0 if and only if the changing activity (i,j) is maintained by the disposition timetable x. The *overall delay* $f(x)$ of a disposition timetable x is

$$f(x) = \sum_{i \in \mathcal{E}_{\text{arr}}} w_i(x_i - \pi_i) + \sum_{a \in \mathcal{A}_{\text{change}}} z^x_a w_a T.$$

approximating the overall delay of passenger arrivals according to the timetable x. Offline delay management assumes that all source delays are known in advance.

Problem (Capacitated Offline Delay-Management). Given a timetable π and a delay state d, find a disposition timetable x for d of minimal overall delay $f(x)$.

The uncapacitated case does not take the limited capacity of the track system into account.

Problem (Uncapacitated Offline Delay-Management). Given a timetable π and a delay state d, find a disposition timetable x for d on $\mathcal{N}' = (\mathcal{E}, \mathcal{A} \setminus \mathcal{A}_{\text{head}})$ of minimal overall delay $f(x)$.

Models for online delay management. In the online case, source delays are not completely known in advance and expectations may change over time. We model this as a process of repeatedly updated knowledge and expectations on future delays. We are led by the following considerations:

- We want to be able to model completely unexpected delays (like accidents) that get known just at the time the respective event is scheduled.
- We want to be able to model changing expectations (like changing plans for working sites).
- The knowledge and expectation on delays are given by a finite sequence $\sigma = (d^1, t^1)$, $(d^2, t^2), \ldots, (d^m, t^m)$ where d^k is a delay state and $t^k \in \mathbb{N}$ the time at which d^k becomes active.
- Between time t^k and t^{k+1} everything happens as expected in state d^k.

From a more technical point of view, we have the request to be able to express realistic, memoryless models that work similar to Markov-chains as well as distributions applied in academics for experiments. The following definitions of *feasible next delay state* and *online strategy for delay management* give the minimal technical restrictions that are necessary to fulfill the above desiderata.

Assume that at time t^1, the current delay state is d^1 and the disposition timetable is x^1. At a point $t^2 > t^1$ in time, the delay state changes to a new state d^2. We assume that all events and delays that were scheduled between t^1 and t^2 happened as planned in x^1. Hence, d^1 and d^2 are equal for all activities (i,j) with $x^1_j < t^2$. A delay state d^2 that guarantees this property is called a *feasible next delay state.*

Definition (Feasible next delay state). Given delay states d^1 and d^2, a timetable x^1 and a point in time $t^2 \in \mathbb{N}$, we call (d^2, t^2) *feasible* for (x^1, d^1) if $d^1_{(i,j)} = d^2_{(i,j)}$ for each activity (i,j) with $x^1_j < t^2$ and $d^1_j = d^2_j$ for each event with $x^1_j < t^2$.

At time t^2, it may turn out that the timetable x^1 is not feasible any more. An *online strategy* (for delay management) adapts a timetable to fit the new delay state. We assume that the re-scheduling is done at time t^2 and all events that were scheduled in x^1 before time t^2 already happened as planned. Hence, only events scheduled (according to x^1) after time t^2 can be rescheduled.

Definition (Online Strategy for Delay Management). Given are a timetable x^1, a time $t^2 \in \mathbb{N}$ and delay states d^1, d^2 such that (d^2, t^2) are feasible for (x^1, d^1). An *online-strategy* for delay management is an algorithm $\mathcal{S}$ that computes a disposition timetable $x^2 = \mathcal{S}(x^1, d^2, t^2)$ for d^2 such that

- $x^2_i = x^1_i$ for each $i \in \mathcal{E}$ with $x^1_i < t^2$ (6.5)
- from $x^1_i \geq t^2$ follows $x^2_i \geq t^2$. (6.6)

Accordingly, we express the current state of the system through the triple (x, d, t) of current disposition timetable x, delay state d and a time t representing the start of d. Then, next delay state and time (d', t') are randomly chosen by a *delay generator.*

Definition (Delay Generator). Let $\mathcal{D}$ be a black-box routine whose input may consist of current disposition timetable x, delay state d, time t, additional random values and values computed in former executions of $\mathcal{D}$. We call $\mathcal{D}$ a *delay generator* if $(d', t') := \mathcal{D}(x, d, t)$ is a feasible next delay state for input (x, d, t) and any additional input.

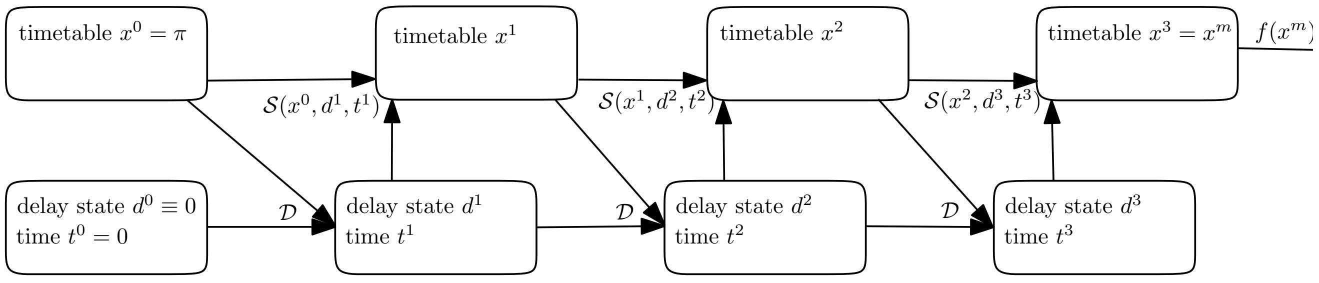

timetable $x^0 = \pi$
timetable x^1
timetable x^2
timetable $x^3 = x^m$
$f(x^m)$
$\mathcal{S}(x^0, d^1, t^1)$
$\mathcal{S}(x^1, d^2, t^2)$
$\mathcal{S}(x^2, d^3, t^3)$
$\mathcal{D}$
$\mathcal{D}$
$\mathcal{D}$
delay state $d^0 \equiv 0$ time $t^0 = 0$
delay state d^1 time t^1
delay state d^2 time t^2
delay state d^3 time t^3

Figure 6.2: Illustration of the delay management process for a delay management strategy $\mathcal{S}$.

Starting with $(x, d, t) = (\pi, 0, 0)$ online delay management first obtains a tuple (d', t') from $\mathcal{D}$. Then, a new disposition timetable x' is computed by a delay management strategy $\mathcal{S}$. The process iteratively repeats until the considered time horizon ends (i.e., it stops, when the roll-out time is reached). The objective function (the delay of all passengers) is computed out of the final disposition timetable.

Definition (Delay Management Process). We are given a delay management strategy $\mathcal{S}$ and a delay generator $\mathcal{D}$. The delay management process $\mathcal{M}$ with respect to $\mathcal{S}$ and $\mathcal{D}$ is the process constructed as follows:

Starting with $x^0 = \pi$, $d^0 \equiv 0$ and $t^0 = 0$, we iteratively obtain $(d^{i+1}, t^{i+1}) := \mathcal{D}(x^i, d^i, t^i)$ and afterwards compute $x^{i+1} = \mathcal{S}(x^i, d^{i+1}, t^{i+1})$. The process ends when $t^{m+1} = \infty$ (and there with the convention $x^{m+1} = x^m$ and $d^{m+1} = d^m$). Let $\sigma = (x^1, d^1, t^1), (x^2, d^2, t^2), \dots, (x^m, d^m, t^m)$ be a realization of $\mathcal{M}$. The *overall delay* of σ is $f(x^m)$.

The process is depicted in Figure 6.2. At a first glance it might look unnecessary that the delay generator also incorporates the current disposition timetable x as an input. The reason is the following: In order to have the next delay state feasible, the delay generator has to assure that events that already happened do not change. In order to know which events already happened it is necessary to know the current disposition timetable x. Online Delay Management aims at finding good average case strategies.

Problem (Capacitated Online Delay Management). Given a delay generator $\mathcal{D}$, find an online strategy $\mathcal{S}$, such that the expected overall delay of the delay process $\mathcal{M}$ of $\mathcal{S}$ and $\mathcal{D}$ is minimal.

Again, the uncapacitated case does not take the limited capacity of the track system into account.

Problem (Uncapacitated Online Delay Management). Uncapacitated Online Delay Management is Capacitated Online Delay Management on $\mathcal{N}' = (\mathcal{E}, \mathcal{A} \setminus \mathcal{A}_{\text{head}})$.

A Delay Generator. Within our experiments we use the delay generator DDA (delays determined in advance) which is described in the following. We assume that delays do *not* depend on the delay management strategy and a final delay state d^{final} randomly is chosen *before* the actual delay management process starts. Note that we still start the process with state $(x^0 = \Pi, d^0 = 0, t^0 = 0)$.

Delays get known over time, directly when they are scheduled to happen, i.e., to compute $(d^{k+1}, t^{k+1}) := \mathcal{D}(x^k, d^k, t^k)$ we search for the next (according to x^k) set of events or activities that are source delayed (according to d^{final}). More formally, let

- $i \in \mathcal{E}$ be such that $d_i^k = 0$, $d_i^{\text{final}} > 0$ and x_i^k is minimal, and

- $(j,w) \in \mathcal{A}$ be such that $d^k_{(j,w)} = 0$, $d^{final}_{(j,w)} > 0$ and x^k_j is minimal

We set $t^{k+1} = \min\{x^k_i, x^k_j\}$. Given arbitrary $a \in \mathcal{E}$, $(v,w) \in \mathcal{A}_{\text{train}}$, the next delay state d^{k+1} is given by

$$d^{k+1}_a = \begin{cases} d^{\text{final}}_a & , x^k_a \le t^{k+1} \text{ or } d^k_a > 0 \\ 0 & , \text{otherwise} \end{cases} \tag{6.7}$$

$$d^{k+1}_{(v,w)} = \begin{cases} d^{\text{final}}_{(v,w)} & , x^k_v \le t^{k+1} \text{ or } d^k_{(v,w)} > 0 \\ 0 & , \text{otherwise.} \end{cases} \tag{6.8}$$

We finish with time $t^{k+1} = \infty$ when $d^k = d^{\text{final}}$. As the delay management process gets deterministic as soon as d^{final} is fixed, we call d^{final} an *instance* of generator DDA.

6.3 Delay Management Strategies

An a-posteriori bound and ILP-approaches. In [SS10] an exact integer programming formulation for the offline problem is given. The formulation is as follows.

$$\min f(x,z,g) := \sum_{i \in \mathcal{E}_{\text{arr}}} w_i(x_i - \pi_i) + \sum_{a \in \mathcal{A}_{\text{change}}} z_a w_a T \tag{6.9}$$

such that

$$\begin{aligned}
x_i &\ge \pi_i + d_i & i &\in \mathcal{E} & (6.10)\\
x_j - x_i &\ge L_{(i,j)} + d_{(i,j)} & (i,j) &\in \mathcal{A}_{\text{train}} & (6.11)\\
Mz_{(i,j)} + x_j &\ge L_{(i,j)} + x_i & (i,j) &\in \mathcal{A}_{\text{change}} & (6.12)\\
Mg_{ij} + x_j &\ge L_{(i,j)} + x_i & (i,j) &\in \mathcal{A}_{\text{head}} & (6.13)\\
g_{ij} + g_{ji} &= 1 & (i,j) &\in \mathcal{A}_{\text{head}} & (6.14)\\
x_i &\in \mathbb{Z}^+ & i &\in \mathcal{E} & (6.15)\\
z_a &\in \{0,1\} & a &\in \mathcal{A}_{\text{change}} & (6.16)\\
g_{ij} &\in \{0,1\} & (i,j) &\in \mathcal{A}_{\text{head}} & (6.17)
\end{aligned}$$

where M is a number ‘big enough’. Detailed explanations on the ILP including a discussion on the size of M can be found in the original work.

When working with delay generator DDA, all source-delays are determined by d^{final} before the delay management process starts. Consider an optimal solution to the offline problem for d^{final}. This solution obviously gives an a-posteriori bound on the corresponding online problem for that instance. The following lemma shows that this bound is tight.

Lemma 21. Let d^{final} be an instance of delay generator DDA. Let x^{opt} be an optimal solution to the offline problem for d^{final}. Then, there is an online strategy $\mathcal{S}$ that generates (on instance d^{final}) a delay management process $\sigma = (x^1, d^0, t^0)$, $(x^2, d^1, t^1), \ldots, (x^m, d^{m-1} = d^{\text{final}}, t^{m-1})$ such that $f(x^{\text{opt}}) = f(x^m)$.

Proof (of Lemma 21). Given d^{final}, the following online strategy does the desired. Let $t_1 = \min\{\pi_i \mid i \in \mathcal{E} \wedge d_i > 0 \text{ or } (i,j) \in \mathcal{A} \wedge d_{(i,j)} > 0\}$ and let $x^0 := \pi$. For $k \ge 1$, we set $\mathcal{S}(x^k, d^{k+1}, t^{k+1})$ to be a timetable x' which is defined by $x'_i = \pi_i$ if $\pi_i < t_1$ and $x'_i = x^{\text{opt}}_i$ otherwise.

Obviously, $\mathcal{S}$ is an online strategy for $k > 0$. Because of $x_i^{\text{opt}} \geq \pi_i$ for any $i \in \mathcal{E}$, we have $w_i(x_i^{\text{opt}} - \pi_i) \geq w_i(x_i' - \pi_i)$. Further, it is $z_{(i,j)}^{x'} = 0$ if $\pi_i < t_1$. (This holds in the case that $\pi_j < t_1$ since then both events are not delayed, and in the case that $\pi_j \geq t_1$ since the transfer can take place if only the departing train has a delay). Accordingly, $z_{(i,j)}^{x'} = z_{(i,j)}^{x^{\text{opt}}}$ if $\pi_i, \pi_j \geq t_1$.

Consequently we have $z_{(i,j)}^{x'} w_a T \leq z_{(i,j)}^{x^{\text{opt}}} w_a T$. Summarizing, it is $f(x') \leq f(x^{\text{opt}})$. To prove $f(x^{\text{opt}}) \geq f(x^m)$, it remains to show that x' is a disposition timetable for each d^k which can easily be done by checking Equations 6.2, 6.3 and 6.4. It is $f(x^{\text{opt}}) \leq f(x^m)$ as x^m must be a disposition timetable for d^m.

By fixing past events and iteratively recomputing the exact offline-solution we can use the ILP as an online strategy:

Strategy (OnlineILP). Given (x^k, d^{k+1}, t^{k+1}), compute $x^{k+1} = \mathcal{S}(x^k, d^{k+1}, t^{k+1})$ by setting $x^{k+1} = x$ for a feasible solution (x, z, g) of

$$\min f(x,z,g) := \sum_{i \in \mathcal{E}_{\text{arr}}} w_i(x_i - \pi_i) + \sum_{a \in \mathcal{A}_{\text{change}}} z_a w_a T \tag{6.18}$$

such that (6.10)-(6.17) and such that

$$x_i = x_i^k \qquad i \in \mathcal{E} \text{ and } x_i^k < t^{k+1} \tag{6.19}$$

$$x_i \geq t^{k+1} \qquad i \in \mathcal{E} \text{ and } x_i^k \geq t^{k+1} \tag{6.20}$$

Ad-Hoc Re-Scheduling. This online strategy iteratively re-schedules the timetable event-by-event. Simple heuristic functions are used to decide time and choice of the next event to schedule. The approach uses only local information when scheduling an event. The resulting disposition timetable strongly depends on the choice of the heuristic functions. Without considering headway constraints the strategy is straightforward: First, order the events that may be influenced by the given delay topologically. Then proceed the events in this order and let each event start as early as possible with respect to the applied heuristic. If headways are regarded, we further refine the topological order and have to obey more restrictions. The computation of the next disposition timetable $x^{k+1} = \mathcal{S}(x^k, d^{k+1}, t^{k+1})$ then works as follows:

For each event i, the time x_i^{k+1} is initialized to x_i^k if $x_i^k < t^{k+1}$. Otherwise we initialize x_i^{k+1} with ∞. During the run of the algorithm, $x_i^{k+1} = \infty$ indicates that event i has not yet been scheduled. For each event i we can compute a wish $\text{time}(i, x^{k+1}, d^{k+1}, t^{k+1})$ for the event to happen. This wish depends on the choice of the already re-scheduled events. It consists of three parts.

$$\begin{aligned}\text{time}(i, x^{k+1}, d^{k+1}, t^{k+1}) := \max\{ \; & \text{time}_{\text{earliest}}(i, x^{k+1}, d^{k+1}, t^{k+1}), \\ & \text{time}_{\text{dm}}(i, x^{k+1}, d^{k+1}), \\ & \text{time}_{\text{top}}(i, x^{k+1})\}\end{aligned}$$

The function $\text{time}_{\text{earliest}}(i, x^{k+1}, d^{k+1}, t^{k+1})$ assures that all technical restrictions are respected and hence makes sure that we gain a (feasible) disposition timetable. The function $\text{time}_{\text{dm}}(i, x^{k+1}, d^{k+1})$ realizes the applied heuristics while $\text{time}_{\text{top}}(i, x^{k+1})$ influences the order in which events are scheduled. All three functions will be described later.

Now, while there is an unscheduled event, we pick an arbitrary unscheduled event i with minimum $\text{time}(i, x^{k+1}, d^{k+1}, t^{k+1})$ and schedule it to happen at that time. Note that scheduling an event i may alter $\text{time}(j, x^{k+1}, d^{k+1}, t^{k+1})$ for other events j. The pseudocode is given as Algorithm 6.1. Each computation step needs only local information. Hence, only small changes are required such that the approach can be applied locally by the trains drivers or by the local disponents at single train stations.

Algorithm 6.1: Ad-Hoc Re-Scheduling

```
input  : time $t^{k+1} \in \mathbb{N}$, old disposition timetable $x^k$, new delay state $d^{k+1}$,
         function time$(\cdot,\cdot,\cdot,\cdot)$
output: new disposition timetable $x^{k+1}$
1 for $i \in \mathcal{E}$ do
2     $x_i^{k+1} \leftarrow \infty$
3     if $x_i^k < t^{k+1}$ then $x_i^{k+1} \leftarrow x_i^k$
4 while there is an event $i$ with $x_i^{k+1} = \infty$ do
5     $i \leftarrow$ choose an arbitrary element in $\{i \in \mathcal{E} \mid x_i^{k+1} = \infty\}$ with minimal
      time$(i, x^{k+1}, d^{k+1}, t^{k+1})$
6     $x_i^{k+1} \leftarrow$ time$(i, x^{k+1}, d^{k+1}, t^{k+1})$
```

Common Parts of all Heuristics. While the function time_{dm} contains the individual part of each heuristic, all applied strategies share the same choice of $\text{time}_{\text{earliest}}$ and time_{top}.

The graph $\mathcal{N}' = (\mathcal{E}, \mathcal{A} \setminus \mathcal{A}_{\text{head}})$ is a acyclic. It turned out that the quality of Ad-Hoc Re-Scheduling significantly increases when we perform the scheduling of the events in topological order of $\mathcal{N}'$. The reason is, that we otherwise loose information when computing time_{dm}. Given an event i, a successor j of i in $\mathcal{N}'$ and the resulting disposition timetable x, that does not necessarily mean that $x_i \leq x_j$ but that we determine the value of x_i before we determine the value of x_j. It further is possible that we determine the value of x_i before we determine the value of x_j even if $x_i > x_j$. The function

$$\text{time}_{\text{top}}(i, x) := \begin{cases} \infty & \text{there is an } (j,i) \in \mathcal{A}_{\text{change}} \text{ with } x_j = \infty \\ 0 & \text{otherwise} \end{cases}$$

ensures that the topological order is respected for changing activities. The task of function $\text{time}_{\text{earliest}}(i, x, d, t)$ is to compute the earliest point in time at which event i can take place with respect to the original timetable π, the predecessing event on the same line, the current time t, the headway constraints of already scheduled events and all known source delays. This also ensures that the topological order is guaranteed for waiting and driving activities. The value

$$\text{earliestNoHead}(i, x, d, t) := \max\left(\{t, \pi_i + d_i\} \cup \{x_j + L_{(j,i)} + d_{(j,i)} \mid (j,i) \in \mathcal{A}_{\text{train}}\}\right)$$

already respects all requirements but the headways. The function nextSlot considers the headway constraints by giving the earliest point in time *at or after* the time $\underline{t}$ at which event i can take place without violating a headway constraint of an already scheduled event, i.e., an event j with $x_j < \infty$.

$$\begin{aligned} \text{nextSlot}(i, x, \underline{t}) := \min\{t \geq \underline{t} \mid \quad & t \notin [x_j, x_j + L_{(j,i)}) \text{ for } (j,i) \in \mathcal{A}_{\text{head}}, x_j < \infty \\ & x_j \notin [t, t + L_{(i,j)}) \text{ for } (i,j) \in \mathcal{A}_{\text{head}}, x_j < \infty\} \end{aligned}$$

Note that this function may be suboptimal in case of $L_{(i,j)} = 0$, but this can easily be handled as a special case. The function $\text{time}_{\text{earliest}}(i, x^{k+1}, d^{k+1}, t^{k+1})$ combines nextSlot and earliestNoHead

$$\text{time}_{\text{earliest}}(i, x, d, t) := \text{nextSlot}(i, x, \text{earliestNoHead}(i, x, d, t))$$

and hence satisfies all technical requirements.

Uniform-Heuristics for Ad-Hoc Re-Scheduling. This class of Ad-hoc Re-Scheduling uses the same decision strategy for all events and does not depend on preprocessed information. The function time_{dm} determines whether a train should wait for a feeder train. In the following, we use the indicator function

$$1_A(y) = \begin{cases} y & , \; y \in A \\ 0 & , \; \text{otherwise} \end{cases}$$

and the convention $\min\{\emptyset\} = 0$. Given a parameter ℓ_i, we define time_{dm} as

$$\text{time}_{\text{dm}}(i, x, d, t) := \max\left\{1_{[0,\ell_i]}(\text{nextSlot}(x, i, x_j + L_{(j,i)} + d_{(j,i)})) \mid (j, i) \in \mathcal{A}_{\text{change}}\right\}.$$

The value of ℓ_i steers how long event i waits for feeder trains. We apply the following choices.

- *Always wait.* Formally: $\ell_i = \infty$
- *Never wait.* Formally: $\ell_i = 0$
- *Wait if enough slack on next activity.*
  Formally: $\ell_i = \min\{\pi_w - L_{(j,w)} - d_{(j,w)} \mid (j, w) \in \mathcal{A}_{\text{train}}\}$
- *Wait y minutes starting from earliest possible departure.*
  Formally: $\ell_i = \text{time}_{\text{earliest}}(i, x, d, t) + y$
- *Wait y minutes starting from original timetable.*
  Formally: $\ell_i = \pi_i + y$

The strategies *always wait*, *never wait*, and *wait y minutes starting from original timetable* have been used before in [KS10], but without respecting headway constraints.

Learning Non-Uniform Heuristics from Simulation. In the non-uniform case, we preprocess some data that is used as a parameter when computing priorities. For each changing activity (i, j) we are given a value $\ell_{(i,j)} \in \mathbb{N}$ meaning that event j should wait at most $\ell_{(i,j)}$ minutes starting from the scheduled timetable in order to maintain (i, j). Accordingly, we have

$$\text{time}_{\text{dm}}(i, x, d) = \max\left\{1_{[0,\pi_i+\ell_{(j,i)}]}(x_j + L_{(j,i)} + d_{(j,i)}) \mid (j, i) \in \mathcal{A}_{\text{change}}\right\}.$$

To learn the values of $\ell_{(i,j)}$ we first generate a number of random delay states. In case of model DDA we use the final delay states d^{final}. Otherwise it is reasonable to obtain final delay states by applying an arbitrary delay management strategy. Afterwards we exactly solve the offline problem on these instances using the ILP. We obtain a sequence (x^1, d^1), …, (x^n, d^n) where disposition timetable x^i is an optimal solution of the random instance d^i. In order to obtain $\ell_{(i,j)}$ for a changing activity (i, j), we compute the multisets

$$\begin{aligned} \text{yes}_{(i,j)} &:= \left\{x_i^k + L_{(i,j)} - \pi_j \mid x_j^k \geq x_i^k + L_{(i,j)}, k \in 1, \ldots, m\right\} \\ \text{no}_{(i,j)} &:= \left\{x_i^k + L_{(i,j)} - \pi_j \mid x_j^k < x_i^k + L_{(i,j)}, k \in 1, \ldots, m\right\}. \end{aligned}$$

These sets contain, for each instance, the value how long event j would have had to wait in order to maintain activity (i, j). The simulated instances $x^1, \ldots, x^m$ are split such that $\text{yes}_{(i,j)}$ represents all instances in which (i, j) actually has been maintained and such that

$\text{no}_{(i,j)}$ represents the remaining instances. We let $\ell_{(i,j)}$ be an arbitrary value that separates $\text{yes}_{(i,j)}$ and $\text{no}_{(i,j)}$ optimally. That is a value $\ell_{(i,j)}$ for which

$$p(\ell_{(i,j)}) := \left|\{v \in \text{yes}_{(i,j)} \mid v > \ell_{(i,j)}\}\right| + \left|\{v \in \text{no}_{(i,j)} \mid v < \ell_{(i,j)}\}\right|$$

is minimal. Experimentally, we made the observation that it usually is possible to split both sets very well, often with $p(\ell_{(i,j)}) = 0$. Further, when considering the quality of the online-phase, the actual choice of $\ell_{(i,j)}$ out of all optimal values only mattered for very small simulation numbers m.

6.4 Experiments

In this section, we present an experimental evaluation of the algorithms described above. Unless stated otherwise experiments are performed on a random sample of size 100 and the preprocessing of the learning heuristics uses 1000 simulation runs.

All experiments are reported as box-and-whiskers plots: There is one entry on the x-axis for every online delay management strategy applied. The y-axis always gives the relative quality of the solutions compared to the respective optimal solutions (i.e., the objective value of the online strategy divided by the objective value of the tight lower bound described in Section 6.3). Bottom and top of the box give the lower and upper quartiles, the band inside a box gives the median and the whiskers give the smallest and highest observations without outliers. We classify any observation outside 1.5 interquartile range of the lower quartile or outside 1.5 interquartile range of the upper quartile as outlier and depict it as a circle.

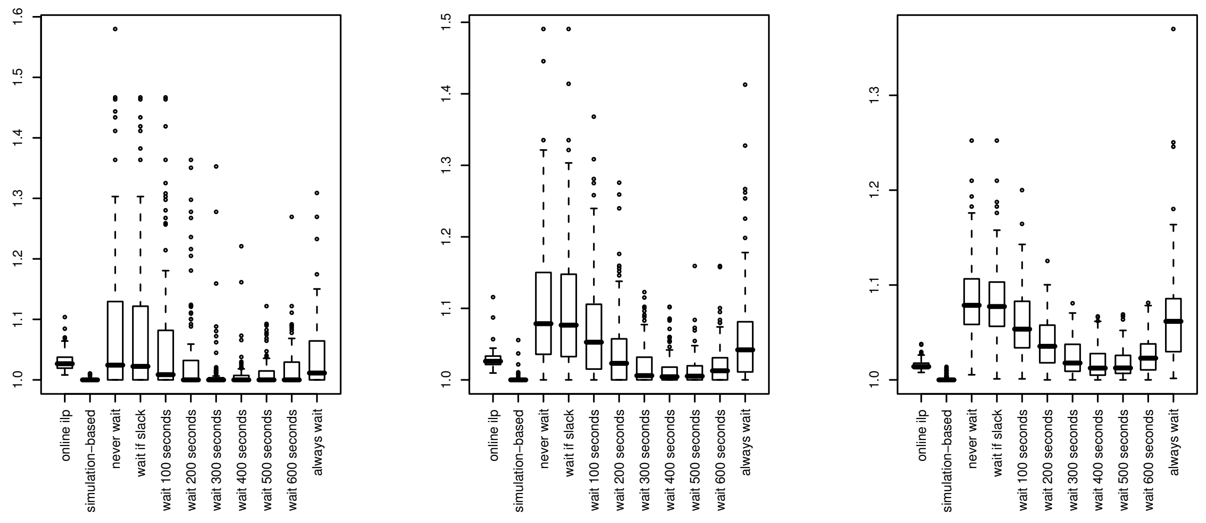
Instances. We work on two different datasets. LinTim is the medium sized real-world based instance that is part of the LinTim-package. See [SS09] for further details. Until now, the dataset does not incorporate headways. The instance Harz is a real-world dataset representing the network of Deutsche Bahn in the Harz region. The instance incorporates headways. Further information on that data can be found in [Sch10]. For both datasets, the *roll-out time* gives the length of the considered time horizon.

We use the delay generator DDA. We generate an instance d^{final} by choosing a given number of activities uniformly at random. Each of them is assigned a random delay, uniformly distributed in the interval [1min, 3min] (scenario weak), [3min, 15min] (scenario medium) or [15min, 18min] (scenario strong). We use the scheme networkname/roll-out-time/number of delays/delay-scenario for identifying particular instances. We add the postfix /noHead for Harz-instances with headway constraints removed.

Small Observations made by Pretests. It turned out that the strategies 'wait y minutes from initial timetable' and 'wait y minutes from earliest possible departure' perform very similar. Hence we only include 'wait y minutes from initial timetable' in our experimental study.

When learning the values $\ell_{(i,j)}$ from simulation, there are many possibilities of how to treat changing activities for which no data has been gained by the simulation. Further, even in case there is data available, $\ell_{(i,j)}$ usually can be chosen out of one (or even more intervals). In our experiments the actual choice of these degrees of freedom only mattered for very small simulation numbers.

Runtime. Our implementation is written in Java using XPRESS as ILP-Solver and the tests were executed on one core of an AMD Opteron 2218, running SUSE Linux 10.3.



Harz/6h/10/medium/noHead Harz/6h/20/medium/noHead Harz/6h/60/medium/noHead

Figure 6.3: Experiments on the Harz instances with headways removed.

The machine is clocked at 2.6 GHz, has 32 GB of RAM and 2 x 1 MB of L2 cache. The program was compiled with Java 1.6. Because of the different focus of the problem we do not report the runtimes in great detail or measure highly accurate. Further, we observed that the applied ILP-solver slightly slowed down with growing number of simulation runs. Hence, the given numbers shall only give an impression on the required runtime.

The average time needed for solving the ILPs mainly depends on size and structure of the network and the roll-out time. The approximate numbers in seconds are < 1 for LinTim/2h, 3 for LinTim/4h, 5 for LinTim/6h, 10 for LinTim/8h, < 1 for Harz/2h, 9 for Harz/4h, 48 for Harz/6h and 234 for Harz/8h. The runtime of Ad-Hoc Re-Scheduling is neglectable. On the instances of this study it usually took less than 1 second to solve one offline problem.

Main Experiments. Our main experiments are given as Figures 6.5 and 6.3. We observe that the OnlineILP and the learning heuristics are both performing very well. Both strategies are always better than the best uniform strategy and often are very close to the optimum.

All experiments on the learning heuristics have been made using 1000 simulation runs. More simulation runs lead to more accurate values of ℓ_a and more activities for which estimations of ℓ_a are available. Accordingly, in Figure 6.4 we see that the quality of the approach strongly depends on the number of simulation runs. Hence, the already good quality of the approach can be further improved by using a longer simulation phase.

There are noticeable, extreme outliers on the weak-instances. The reason is simple: On these instances, the part of the objective function representing delayed arrivals in stations is so small that any wrong wait-decision leads to these extreme values.

We now have a short look at the Harz/noHead-instances. Comparing the strategies against each other, the situation here is similar to the LinTim-instances. The OnlineILP performs slightly worse on Harz/noHead and there is a significantly smaller gap between the always-wait and the never-wait strategy. A big difference lies in the absolute approximation-ratios. These are much better for all strategies on Harz/noHead. We interpret that as follows: There is much less influence of the applied strategy on the objective function and hence, less space for improvement on the Harz/noHead-instances.

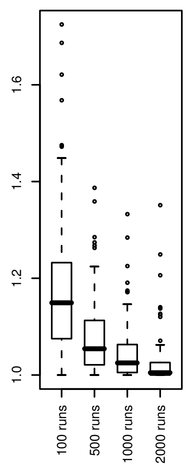


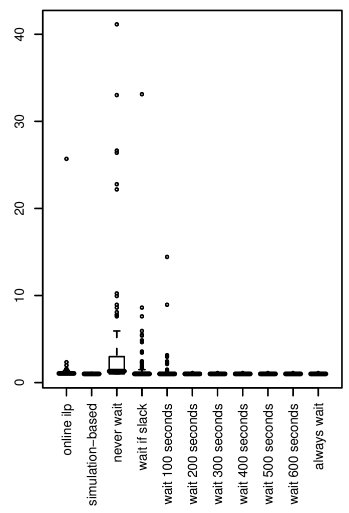
Figure 6.4: Performance of Simulation-Based Ad-Hoc Re-Scheduling on LinTim/8h/20/medium for different numbers of simulation runs.

Influence of the Rollout Time. We also checked if the roll-out time has influence on the performance of the applied strategies. It turned out that this influence is small and the strategies' performance is quite robust with respect to roll-out time. As a small impact of the roll-out time, good solutions on longer roll-out times tend to have slightly less waiting decisions. This is easily explainable as in this case waiting decisions can propagate further into the future. Some experiments on the roll-out time can be seen in Figure 6.6.

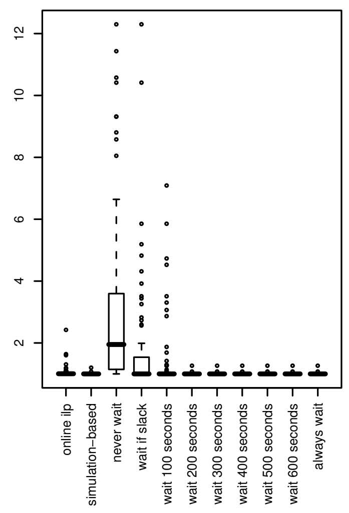
Robustness Issues. The preprocessing phase of our learning heuristics makes use of knowledge on the underlying delay distribution. Hence we tested the robustness of the approach with respect to deviations in the delay distribution. To that end, we trained our strategy for the delay scenario 20/medium and used the following scenarios in the online phase: 10/medium, 60/medium, 20/weak, 20/strong. The results are given in Figure 6.7. The approach shows to be robust and the solution quality only decreases little. However the scenario strong deviates heavy enough from medium to effect in a noticeable decrease in solution quality. This suggests the following modus operandi for practical application: Perform preprocessing for a 'standard' and one or two 'extreme' scenarios (incorporating more or stronger delays). In the online phase apply the standard values unless an extreme situation is detected. In case of identifying an extreme situation within the online phase, it is technically unproblematic to immediately switch to the preprocessed data for the according situation.

Towards Learning Headways. The experiments seen so far were based on the uncapacitated model, i.e., aimed at solving Problem 2. However, Ad-hoc Re-Scheduling also computes feasible solutions in the presence of headway constraints. The only realistic dataset incorporating headways we have access to are the Harz-instances. We did some preliminary tests on these instances. Often all strategies performed very similar and the results were considerably worse than the results obtained for the uncapacitated case. The results of the 'best' instance we tested are given in Figure 6.8.

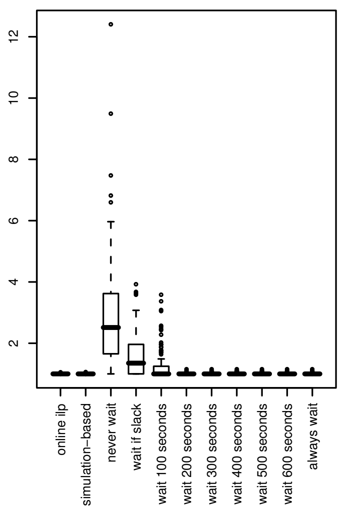
We did some further diagnostics on the Harz-instances and came to the following conclusion. On the Harz-instance, the influence of wait-depart decisions on the objective function is rather small even when not considering headway constraints (see Figure 6.3).



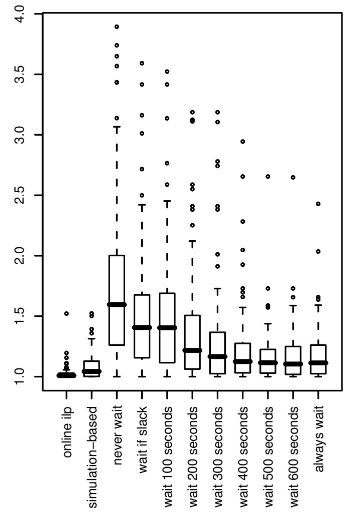
LinTim/8h/10/weak



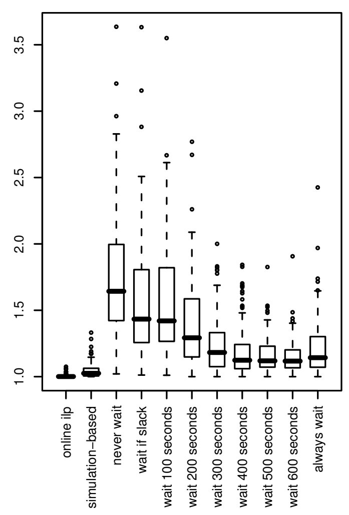
LinTim/8h/20/weak



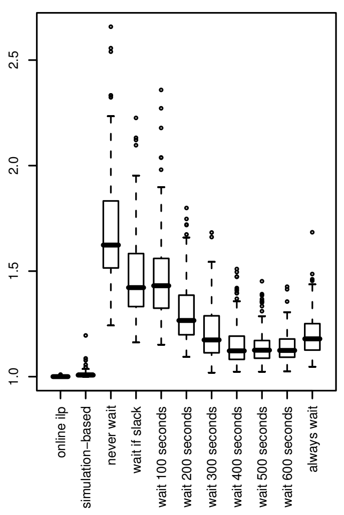
LinTim/8h/60/weak



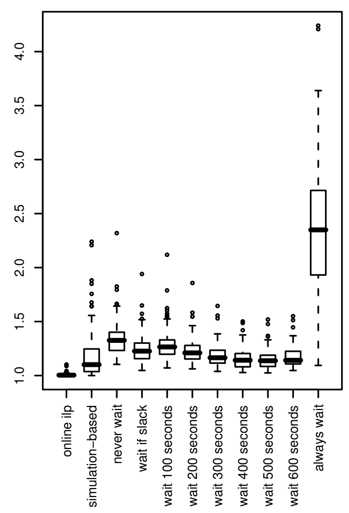
LinTim/8h/10/medium



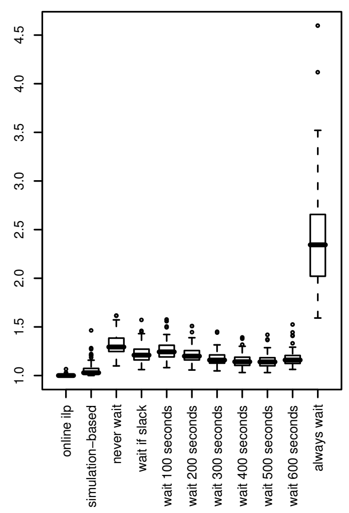
LinTim/8h/20/medium



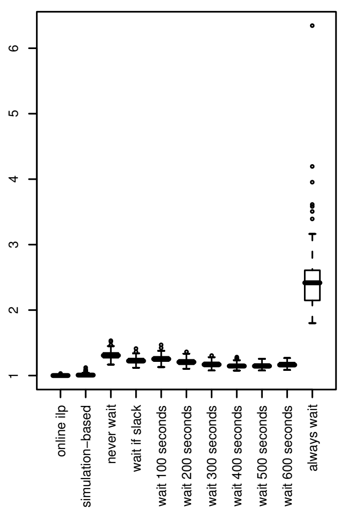
LinTim/8h/60/medium



LinTim/8h/10/strong

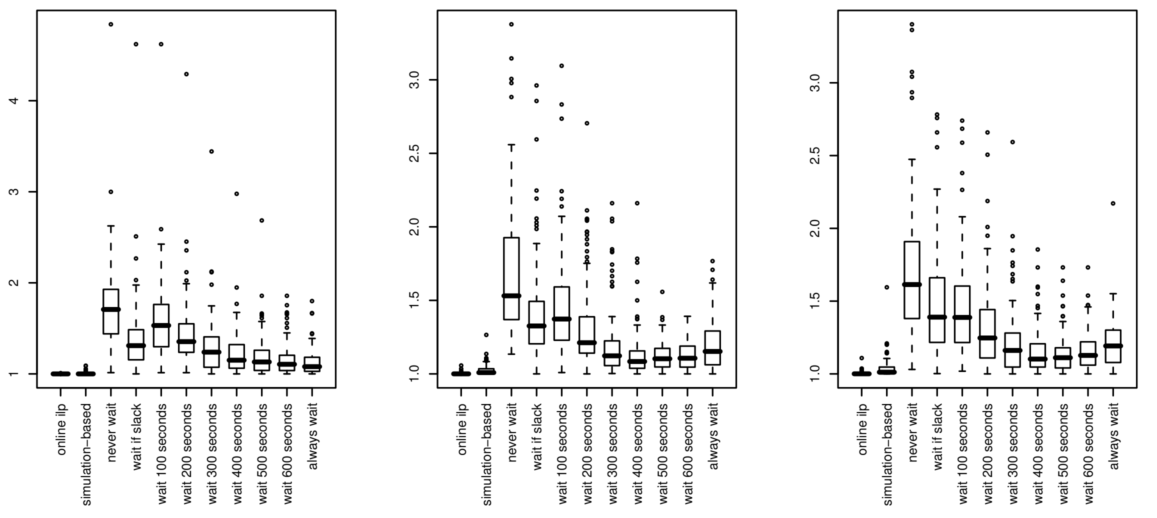


LinTim/8h/20/strong



LinTim/8h/60/strong

Figure 6.5: Main experiments on the LinTim-instances. The number of simulation runs for Ad-Hoc Re-Scheduling is always 1000.



LinTim/2h/20/medium LinTim/4h/20/medium LinTim6h/20/medium

Figure 6.6: Experiments on different rollout times.

When we additionally consider headway constraints on the Harz-instances, this little influence is compensated by the impact of the priority decisions (that determine which headway to favor). Ad-Hoc Re-Scheduling handles headway constraints in a first-come first-served manner, which of course is not optimal. In [Sch10] it is shown that this strategy works well for small delays but is not suitable for large delays. In this chapter we focus on wait-depart decisions, hence we did not study this issue in detail. However, we checked if the learning heuristics can also be applied to learn priority-decisions.

To that end, we performed simulation runs by optimally solving random instances. For a given solution x and a headway constraint (i,j), we define $h_i^k := \max\{\pi_i + d_i, \max\{x_w^k + L_{(w,i)} + d_{(w,i)} | (w,i) \in \mathcal{A}_{\text{train}}\}\}$ and split the multi-sets

$$\text{prefer}_{i<j} := \left\{ h_i^k - h_j^k \mid x_i^k < x_j^k, k \in 1,\dots,m \right\}$$
$$\text{prefer}_{i>j} := \left\{ h_i^k - h_j^k \mid x_i^k > x_j^k, k \in 1,\dots,m \right\}$$

as described the previous section. Again, both sets could be separated very well by a single value. This is a strong hint for the applicability of the learning heuristics also on priority decisions.

Insights in the Structure of Good Solutions. While the delay management problem allows complex interactions between different regions in the underlying network, our results suggest that the impact of wait-depart decisions is astonishingly local on typical instances. Even more surprising, optimal solutions mostly stick to the following simple decision rule: Maintain changing activity (i,j) if j has to wait no more than time $\ell_{(i,j)}$ on event i (for some value $\ell_{(i,j)}$ which has to be chosen extremely carefully). As we worked on datasets of different origin we expect that our results can be generalized to a larger class of real-world instances.

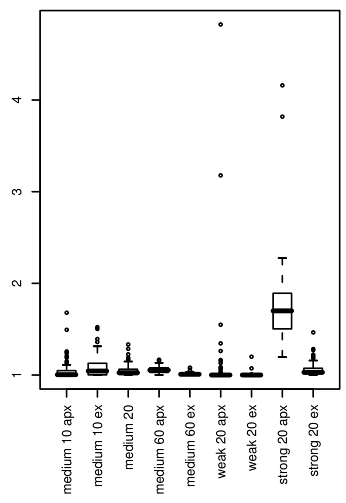


Figure 6.7: Performance of Simulation-Based Ad-Hoc Re-Scheduling on LinTim/8h/X where X is given in the respective column. 'apx' indicates preprocessing on LinTim/8h/20/medium, 'ex' on instance X.

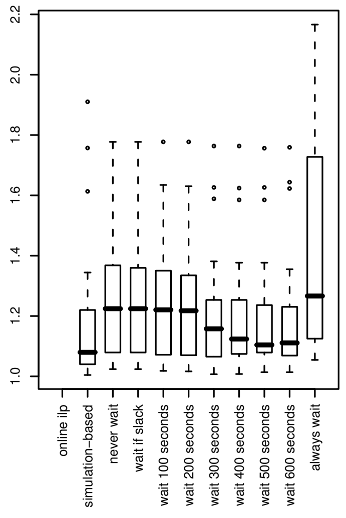


Figure 6.8: Harz/4h/20/medium no results are given for online ilp as it was not possible to compute this strategy in reasonable time.

6.5 Conclusion

This chapter concentrates on wait-depart decisions for online delay management. We enhance the offline model described in [SS10] to the online case and only introduce additional restrictions that are required to assure that the delay management process is well-defined. Hence, we gain a generic model that is able to model a wide range of scenarios. Consequently, the considered online strategies only use fundamental information and can be applied to a broad number of different settings.

We state three approaches for solving the problem: An ILP-based approach (that is based on the corresponding offline-ILP), a class of simple 'rule of thumb' strategies and a learning heuristics that identifies the structure of good solutions using a simulation-based preprocessing phase.

All strategies are evaluated in an extensive experimental study on real-world and real-world based instances. The ILP-based and the learning heuristics perform very well, usually resulting in near-optimal solutions. An additional advantage of the learning heuristics is its simplicity, robustness, and speed. This leads to better practicability in real-world applications. Furthermore, our experiments indicate that the learning-based approach can be enhanced to also suggest good priority decisions in case of capacity constraints.

Finally, we gain interesting new insights into the structure of good solutions: Firstly, single wait-depart decisions have in real-world railway data a surprisingly local impact on the solution. Secondly, optimal solutions mostly stick to a decision rule which is astonishingly simple applicable once its parameters have been identified.

Chapter 7

Conclusion

Summary. In this thesis we addressed four problems in the area of shortest-paths computation and algorithms for infrastructure networks. The character of the considered problems differs considerably. Issues treated in this work range from completely theoretical considerations to purely experimental efforts.

We theoretically analyzed the preprocessing phases of recent algorithms for point-to-point shortest-paths computation. These leave open some degrees of freedom which we proved to be NP-hard to fill. While the hardness of several of these techniques has been conjectured, the work in this thesis thoroughly paves the way for further theoretical work on the given problems. Probably, the most desireable future results are preprocessing strategies with quality guarantees and approximation algorithms. Another interesting question is the following: Given unrestricted preprocessing time and space, how good can a speedup technique actually get? While some techniques can, in some way, encode all-pairs shortest-paths in the preprocessed data, the situation is not so clear for others. In this context, we gave a lower bound for Contraction Hierarchies.

The shortcut problem is a graph-augmentation problem arising from a technique applied in route-planning algorithms. While the problem statement originates from algorithm engineering approaches, we mainly treated it as a theoretical problem that is interesting on its own right. We studied the computational complexity and proposed exact and approximative algorithms.

Dynamic problems are problems in which the input-data changes with time. Most real-world problems are inherently dynamic. We considered the batch-dynamic single-source shortest-paths problem on graphs with positive edge weights. There exist several algorithms for that task. Some of them have been developed with respect to theoretical aspects, others concentrate on practical performance. For none of them, a proof exists for being generally faster than Dijkstra's algorithm. Until now, there was little experimental knowledge on these algorithms and the big picture was unclear. We presented new algorithms for the problem and conducted an extensive experimental study including all existing algorithms and a wide set of different instances. The most surprising result was the astonishing level of data dependency of both the algorithms and the problem itself. This shows, reaching even beyond the field of shortest-paths computation, that the experimental assessment of an algorithm's performance cannot be done by simply testing it on some few instances of similar origin, properties and size.

Finally, we worked on online delay management for passenger-oriented railway. We enhanced offline models as to serve the online case and focused on simple and robust delay management strategies. The main point for optimization in our model are wait-depart decisions, which settle the question if a follow-up train should wait in order to enable changing from delayed feeder trains. We briefly consider also headway constraints, which model the limited capacity of the track system. The results show that a simple, learning-based approach that relies on simulation achieves near-optimal solutions. The development of this strategy led to the insight that optimal solutions mostly stick to an astonishingly simple and local decision rule. As we worked on datasets of different origins, we expect that our results can be generalized to a larger class of real-world instances. The learning-based approach focuses on wait-depart decisions. A reasonable next step is to generalize the approach for also learning headway decisions. First experiments confirm this to be promising.

Outlook. The classical approach of algorithm design either completely goes without experiments or simply uses them to confirm expectations deduced from theory. The wide availability of meaningful real-world data strongly influenced algorithmics. This is accompanied by an increasing complexity of modern hardware, effecting in less predictability of an algorithm's performance and worse accuracy of the standard models like the REAL-RAM. Consequently, algorithmics slowly changes from a theoretical science to a theoretical *and* experimental science. The 'invention' of and increasing attention to algorithm engineering is a direct effect.

Working in the field of shortest-paths computation was a great pleasure, as it opened the opportunity to see part of the future of algorithm design. The search for helpful patterns or properties in real-world data, the integration of experiments as a central part of algorithm design and the re-orientation from beautiful yet simple theoretical problems towards also considering complex, sometimes technical and ugly but important real-world applications will further gain in importance.

From the example of shortest-paths computation we can also learn that the algorithm engineering cycle can be a longsome process. While there often is a fast-rotating smaller cycle of experiments, evaluation and heuristic improvement, the overall process can be slow. Concerning the overall cycle that also contains theory and deeper insights, advance is only made step-by-step and distributed all over the community. Insights made by one group are reused and improved by another, techniques are combined or simplified again. It took many years to develop the techniques currently up-to-date and the theoretical work just started.

On the other hand, the path towards an experimental science is not finished yet. I want to point out two problem fields that still lack progress. Firstly, reproducibility and clarity in description are obligatory in any experimental science. This is sometimes hard to cope with in algorithmics as data may be confidential or implementations may incorporate a vast number of technical details, many of them important. Although there is big effort to ensure reproducibility, there is still no generally accepted, easy-to-use standard way for doing so. Secondly, combining engineering with theory is rather uncommon and the main merits of algorithm engineering still are of purely experimental nature. One reason could be that theory for algorithm engineering often turns out to contain many technicalities. Perhaps a better 'technical toolbox' for such work could help to overcome that problem.

The demand for efficient algorithms for real-world applications will further grow. Algorithm engineering is an interesting field of research that still undergoes changes and that will play a key role in satisfying this demand.

Appendix A

Extended Tables

	LUX	NLD	DEU	RAIL	CAIDA	AS-HOP	AS-RAN	GRID100	GRID300	UN-HOP	UN-EU
FMN	42	1504	29087	151	22702	1624	2182	25	142	327	36
SWSF-FP	112	3759	65404	366	12429	416	691	59	351	1613	31
tunSWSF-FP	152	5140	84873	562	16406	893	3442	105	598	2436	186
tunSWSF-NAR	147	3354	70245	215	9306	614	695	94	523	748	129
tunSWSF-RR	118	3798	66068	412	26093	2148	3766	74	430	2096	102
RR	155	4666	74857	510	34586	2599	4057	103	568	2519	137
RR dag	149	4340	71293	498	36801	2752	3882	95	534	2801	116
Nar-1st Heap	250	5192	97532	533	6438	410	304	146	821	1117	153
Nar-2nd Heap	274	5189	97035	554	6385	411	303	155	857	1063	110
Nar-1st BF	284	5335	100944	357	6578	417	305	138	784	1176	20
Nar-2nd BF	275	4636	99886	467	6494	411	304	156	871	1061	57
$\Delta(G,U)$	130.42	140.52	70.72	30.68	0.21	0.41	0.74	59	113	0.01	93
exp. speedup	236	6762	62549	986	inf	inf	inf	169	804	inf	163

Table A.1: Speedups of single edge updates - extended table.

	LUX	NLD	DEU	RAIL	CAIDA	AS-HOP	AS-RAN	GRID100	GRID300	UN-HOP	UN-EU
FMN	62	1884	690	339	21427	1444	3592	50	304	257	681
SWSF-FP	142	4855	1490	742	12218	347	989	99	737	439	543
tunSWSF-FP	190	6202	1986	1049	15806	767	2151	165	1185	873	1676
tunSWSF-NAR	154	3128	2326	369	13137	462	769	104	618	428	525
tunSWSF-RR	143	5066	1495	763	27671	1798	4201	120	881	1043	1147
RR	322	6172	4619	1139	34537	2427	5046	221	1336	1457	1714
RR dag	307	6250	4479	1078	30886	2237	4680	205	1254	1399	1627
Nar-1st Heap	263	6349	3930	748	9373	274	782	202	1141	897	1234
Nar-2nd Heap	189	5106	2686	642	9183	275	766	143	877	759	1101
Nar-1st BF	55	2504	246	383	9770	280	800	23	105	587	560
Nar-2nd BF	26	1219	148	237	9805	282	792	10	51	394	331
$\Delta(G,U)$	52	59	977	7	0.23	0.22	0.12	19	35	2	2
exp. speedup	589	16045	4486	4930	inf	inf	inf	556	2647	7500	15000

Table A.2: Speedups of single edge updates, edge failure and recovery - extended table.

	LUX	NLD	DEU	RAIL	CAIDA	AS-HOP	AS-RAN	GRID100	GRID300	UN-HOP	UN-EU
FMN	2	12	184	7	1571	69	49	1	4	18	1
ittun SWSF-FP	6	56	784	35	1183	49	136	5	19	222	7
SWSF-FP	5	40	534	18	773	28	9	3	11	101	1
tunSWSF-FP	7	58	767	35	1207	49	139	6	19	236	7
tunSWSF-NAR	7	50	839	15	636	21	12	5	16	29	6
tunSWSF-RR	5	40	595	25	3913	409	183	4	14	400	4
RR	7	47	783	32	5501	592	236	6	18	618	5
RR dag	7	47	764	30	5556	627	220	5	17	658	4
Nar-1st Heap	12	73		31	738	9	33	9	25	71	6
Nar-2nd Heap	13	70		33	721	9	33	9	27	65	4
Nar-1st BF	12	63		19	753	9	33	8	17	73	1
Nar-2nd BF	13	73		29	729	9	33	9	27	64	2
itNar-1st Heap	12	72		30	728	9	33	8	25	70	6
itNar-2nd Heap	12	73		31	709	9	33	9	27	63	4
$\delta(G,U)$	3081	10833	5414	727	8	6	90	1420	4160	4	2612
$\Delta(G,U)$	2886	10468	5414	723	8	6	90	1304	4059	4	2340
exp. speedup	11	90	809	41	23864	4652	310	8	22	5000	6

Table A.3: Speedups of experiments with 25 edges chosen uniformly at random.

	AS-HOP			AS-RAN			CAIDA		
degree	1-10	10-100	100-500	1-10	10-100	100-500	1-10	10-100	100-500
FMN	784	173	23	1368	129	3	7824	2284	185
ittun SWSF-FP	912	228	26	1320	235	11	12874	4212	382
SWSF-FP	273	68	8	389	28	1	9651	2203	128
tunSWSF-FP	967	250	24	1417	252	15	14042	4693	405
tunSWSF-NA	407	92	9	410	50	4	9785	2187	122
tunSWSF-RR	1272	528	130	2475	433	21	12395	6839	969
RR	1438	576	142	2623	490	17	13915	7075	1163
RR dag	1377	550	116	2351	462	16	12723	6228	938
Nar-1st Heap	53	21	9	86	59	16	4315	761	75
Nar-2nd Heap	53	21	9	86	58	15	4294	751	73
Nar-1st BF	53	21	9	87	58	13	4386	762	74
Nar-2nd BF	53	21	9	86	53	9	4329	747	71
itNar-1st Heap	52	18	6	71	30	8	4060	573	63
itNar-2nd Heap	51	18	6	71	29	7	3976	564	60
$\delta(G,U)$	1.26	12.16	82.54	1.47	45.01	1365.85	1.97	7.4	90.99
$\Delta(G,U)$	1.07	8.59	71.28	1.1	34.6	712.3	1.45	5.7	85.73
exp. speedup	27909	3489	393	27909	821	86	190914	38183	2246

Table A.4: Speedups of experiments with node failure and recovery on INTERNET-instances - extended table.

metric	hop			euclidean			energy		
average degree	7	10	15	7	10	15	7	10	15
FMN	30	40	55	27	21	24	12	14	20
ittun SWSF-FP	238	398	485	116	95	98	56	66	91
SWSF-FP	128	214	236	60	32	36	28	24	22
tunSWSF-FP	260	462	561	158	115	141	75	86	110
tunSWSF-NAR	106	116	147	101	77	97	57	61	67
tunSWSF-RR	223	395	527	105	75	89	49	54	67
RR	289	504	628	111	91	106	55	63	84
RR dag	255	422	561	102	81	94	52	58	76
Nar-1st Heap	70	87	131	84	62	111	52	62	74
Nar-2nd Heap	62	76	120	73	55	94	44	52	62
Nar-1st BF	49	63	109	7	2	8	2	2	3
Nar-2nd BF	32	44	87	5	1	4	1	1	2
itNar-1st Heap	55	71	100	64	50	66	36	46	52
itNar-2nd Heap	50	64	93	56	44	52	31	39	44
$\delta(G, U)$	19	8	6	86	107	99	194	174	132
$\Delta(G, U)$	18	7	5	54	79	55	128	119	98
exp. speedup	833	2500	3750	283	190	273	117	126	153

Table A.5: Speedups of experiments with node failure and recovery on UNIT DISK-instances - extended table.

	GRID			LUX			NLD			DEU		
# edge updates	10	20	30	5	10	20	10	20	30	10	20	30
FMN	3	2	1	4	2	1	11	5	2	185	30	7
ittun SWSF-FP	15	9	5	15	7	2	39	17	6	755	100	23
SWSF-FP	13	10	6	15	9	5	75	32	12	873	173	40
tunSWSF-FP	23	16	9	20	12	6	107	44	17	1210	235	55
tunSWSF-NAR	22	16	9	22	13	7	107	41	17	1402	342	79
tunSWSF-RR	16	12	7	15	9	5	72	31	12	957	181	42
RR	17	10	5	20	9	3	43	18	6	924	149	36
RR dag	16	9	5	19	8	3	42	17	6	859	143	34
Nar-1st Heap	16	9	5	20	10	4	37	15	5	1120	196	35
Nar-2nd Heap	16	9	5	22	11	4	37	14	5	1153	197	36
Nar-1st BF	2	1	1	10	7	2	12	7	2	524	71	11
Nar-2nd BF	13	7	4	20	10	5	35	14	5	1041	155	29
itNar-1st Heap	19	12	6	24	12	4	57	24	8	1231	219	54
itNar-2nd Heap	22	13	6	28	13	5	55	23	8	1421	249	67
$\delta(G, U)$	4367	7909	15552	1178	3052	8616	12910	32088	93725	7885	39260	142191
$\Delta(G, U)$	2591	3564	6412	821	1366	2567	4134	10899	26153	3884	13701	51252
exp. speedup	35	25	14	37	22	12	229	87	36	1127	320	85

Table A.6: Speedups of experiments with traffic jam updates - extended table.

	RAIL	LUX
FMN	271	15
ittun SWSF-FP	748	55
SWSF-FP	638	50
tunSWSF-FP	892	66
tunSWSF-NAR	535	77
tunSWSF-RR	556	51
RR	697	75
RR dag	638	72
Nar-1st Heap	312	74
Nar-2nd Heap	295	65
Nar-1st BF	302	22
Nar-2nd BF	261	12
itNar-1st Heap	263	69
itNar-2nd Heap	250	57
$\delta(G, U)$	13.31	281.6
$\Delta(G, U)$	9.5	227.3
exp. speedup	3286	135

Table A.7: Speedups of experiments with node-failure-and-recovery updates on additional graphs.

Appendix B

Review on Complexity Results

In this section we shortly report complexity results on the algorithms described in Section 5.3 which are taken out of the original works.

FMN. The algorithm FMN has worst-case runtime in $O(k \log |V|)$ if we consider only weight updates of edges and the underlying graph $G = (V, E, \text{len})$ has a k-bounded accounting function.

In case we consider a sequence of updates including insertions and deletions, then each output update requires $O(k \log |V|)$ amortized time when the underlying graph $G = (V, E, \text{len})$, augmented by all edges that get inserted during the updates, has a k-bounded accounting function.

SWSF-FP. The algorithm SWSF-FP requires time in $O(\parallel \delta \parallel (\log \parallel \delta \parallel + M_\delta))$ where we use the following notation: A vertex is said to be *modified* if it is not the source and if it is the target node of an updated edge. A vertex is said to be *affected* if its distance changes. A node is said to be *changed* if it is modified or affected. With $|\delta|$ we denote the number of changed nodes. With $\parallel \delta \parallel$ we denote the number of changed nodes plus the number of all edges adjacent to a changed node. Finally, M_δ denotes the time required to solve the Bellman Ford Equations for a changed node.

RR. The algorithm RR processes a single edge update in time $O(\parallel \delta \parallel + |\delta| \log |\delta|)$ where we use the same notation as for the algorithm SWSF-FP except for the term modified. A vertex is said to be *modified* if it is adjacent to an updated edge.

Narváez. The algorithms of the Narváez-Framework have the following runtimes:

Queue Type	First Variant	Second Variant
FIFO	$O(D_{max} \cdot \delta_d^2)$	$O(D_{max} \cdot \delta_d^3)$
D'Esopo Pape	no polynomial upper bound	no polynomial upper bound
PQ: Linear List	$O(\delta_d^2 + D_{max} \cdot \delta_d)$	$O(\delta_{pd}\delta_d + \gamma \cdot D_{max} \cdot \delta_d)$
PQ: Binary Heap	$O(D_{max} \cdot \delta_d \cdot \log \delta_d)$	$O(\gamma \cdot D_{max} \cdot \delta_d \cdot \log \delta_d)$
PQ: Fibonacci Heap	$O(\delta_d \cdot \log \delta_d + D_{max} \cdot \delta_d)$	$O(\delta_d \cdot \log \delta_d + \gamma \cdot D_{max} \cdot \delta_d)$

Symbols mean the following: δ_d is the minimum number of nodes whose distance *or* parent attribute (or both) must change (between in- and output independently of the algorithm applied). The value δ_{pd} is the minimum number of nodes whose distance *and* parent attributes must change (between in- and output independently of the algorithm applied). D_{max} denotes the maximum node degree. Finally, γ denotes the *redundancy factor*, which represents the average time that each node is visited by the algorithm.

Deutsche Zusammenfassung (German Summary)

Algorithm Engineering ist eine moderne Methode des Algorithmenentwurfs. Der Kern dieses Ansatzes ist ein Kreislauf aus Entwurf, theoretischer Analyse, Implementierung und experimenteller Bewertung von praktikablen Algorithmen. Die experimentelle Bewertung soll hierbei Rückschlüsse auf Entwurf und Theorie erlauben und den Kreislauf neu anstoßen. Theoretische Betrachtungen sollen Entwicklung, Verbesserung und Verständnis effizienter Algorithmen fördern. Ein besonderer Fokus liegt häufig auf der Arbeit mit Realweltdaten und Realweltanwendungen.

Diese Arbeit wird von der Idee des Algorithm Engineering geleitet und gliedert sich in vier Teile aus dem Bereich der Kürzeste-Wege-Suche und der Algorithmen für Infrastrukturnetzwerke. Für jedes Problemfeld wird ein jeweils sinnvoller nächster Schritt im Sinne des Algorithm Engineering bestimmt und durchgeführt. Die einzelnen Problemfelder, sowie die entsprechenden Ergebnisse werden im Folgenden näher erläutert.

Theoretische Betrachtung von Punkt-Zu-Punkt-Kürzeste-Wege-Techniken. Im Jahr 1999 veröffentlichten Schulz, Wagner und Weihe die erste Arbeit zur schnellen Berechnung von kürzesten Wegen zwischen beliebigen Punktepaaren in einem (Bahn-)Netzwerk unter Ausnutzung einer Vorberechnungsphase [SWW99]. Darauf folgte ein beachtlicher Wettlauf vieler Arbeitsgruppen um die schnellste Technik zur exakten Punkt-zu-Punkt-Berechnung von kürzesten Wegen in großen Netzwerken. Die resultierenden Ansätze sind meist maßgeschneidert für Straßennetzwerke und auf diesen bis zu 3.000.000 mal schneller als Dijkstra's Algorithmus. Eine Besonderheit ist die Verfügbarkeit von aussagekräftigen Realweltdaten: Graphen, die das Straßennetzwerk der USA und Europas abbilden, sind für wissenschaftliche Zwecke verfügbar. Neben einer großen Anzahl an Arbeiten zum bisherigen Kernproblem der „Punkt-Zu-Punkt-Kürzeste-Wege-Berechnung in statischen Netzwerken“ gibt es mittlerweile auch eine Reihe von Arbeiten zu erweiterten Problemen wie zeitabhängiger Routenplanung, Routenplanung unter Berücksichtigung verschiedener Verkehrsmittel und dem gleichzeitigen Berechnen von mehreren kürzesten Wegen.

Das Themengebiet wird heute als eines der Paradebeispiele für Algorithm Engineering angesehen. Die sehr guten experimentellen Ergebnisse sind allerdings bis heute noch nicht theoretisch fundiert. Diese Arbeit möchte durch folgenden Ansatz zu einer theoretischen Basis beitragen.

Fast alle Techniken weisen Freiheitsgrade in ihrer Vorberechnungsphase auf. Typische Beispiele sind verschiedene Möglichkeiten, einen Graphen zu partitionieren oder zusätzliche Kanten in ein Netzwerk einzufügen. Diese Freiheitsgrade werden in der Praxis rein heuristisch gefüllt. Über die Komplexität, dies möglichst gut zu tun, ist bis jetzt nichts bekannt. Um dies zu ändern wurden alle relevanten Techniken in einem einheitlichen Rahmen modelliert. Als Zielfunktion wurde der durchschnittliche Suchraum einer Start-Ziel-Anfrage gewählt, die Größe von vorberechneten Daten soll hierbei beschränkt sein. Es stellt sich heraus, dass es für jede Technik NP-schwer ist, den jeweiligen Freiheitsgrad optimal auszunutzen. Dies rechtfertigt den Einsatz von Heuristiken.

Das „Shortcut Problem“. Dieses Problem beschäftigt sich mit dem Hinzufügen von *Abkürzungen* zu einem Graphen. Eine Abkürzung ist eine Kante, die einem Graphen hinzugefügt wird und deren Länge genau die Distanz von Start- zu Zielknoten ist. Der Einsatz von Abkürzungen ist eine der verbreitesten und effektivsten Techniken zur Punkt-Zu-Punkt-Kürzeste-Wege-Suche. Die Strategie, nach der Abkürzungen eingefügt und ausgenutzt werden, variiert aber. Unabhängig von einer bestimmten Technik untersuchen wir folgendes Graphaugmentierungsproblem:

Gegeben sei ein Graph G, zu dem k Abkürzungen hinzugefügt werden sollen. Danach werden zwei Knoten s und t zufällig und gleichverteilt aus G ausgewählt und ein kantenminimaler kürzester s-t Weg P berechnet. Wie können die Abkürzungen hinzugefügt werden, so dass die erwartete Anzahl $|P|$ von Kanten in P minimal wird? Zusätzlich wird eine Variante betrachtet, bei der der erwartete Nutzen vorgegeben ist und die Anzahl der dafür benötigten Abkürzungen zu minimieren ist.

Die wesentlichen Ergebnisse dieses Teilbereichs sind: Beweis der NP-Schwere beider Varianten, sowie Beweis der NP-Schwere für bestimmte Approximationsaufgaben. Außerdem exakte, ILP-basierte Ansätze und ein Approximationsalgorithmus für Graphen in denen kürzeste Wege eindeutig sind.

Experimentelle Studie zu dynamischen Kürzeste-Wege-Algorithmen. Ein dynamischer Graph ist ein Graph, der sich mit der Zeit ändert. Erlaubte Operationen sind Löschen und Hinzufügen von Kanten oder Knoten, sowie die Neuwahl von Kantengewichten. Das Problem besteht darin, für einen dynamischen Graphen G zu jedem Zeitschritt die Distanzen von einem vorher gewählten Quellknoten zu allen anderen Knoten in G zu unterhalten. Zu Beginn eines neuen Zeitschrittes soll die Distanztabelle schneller angepasst werden, als eine komplette Neuberechnung benötigen würde.

Für das Problem gibt es bereits Ansätze von Ramalingam und Reps [RR96b, RR96a], Frigioni et al. [FMSN00] und Narváez et al. [NST00] die auch schon durch einige kleine Experimente praktisch evaluiert wurden. Dieser Teil der Arbeit evaluiert die bestehenden Algorithmen auf einer breiteren Datenbasis und stellt außerdem eine Verbesserung für einen der Ansätze von Ramalingam und Reps vor. In Realweltdaten ändern sich häufig mehrere Kanten während eines Zeitschrittes und diese Kanten liegen oft nahe beieinander. Dieses Szenario wird gesondert betrachtet, die Ergebnisse sind wie folgt:

- Auf dem benutzen Testfeld hängt die Güte der Algorithmen in überraschend hohem Maße von den zugrundeliegenden Instanzen ab. Dies schränkt die Aussagekraft der bestehenden Evaluationen ein, die immer nur auf einer sehr eingeschänkten Datenbasis arbeiten.

- Bisher gab es keine experimentellen Ergebnisse, die mehrere Änderungen pro Zeitschritt betrachten. Es zeigt sich, dass es sinvoll ist, diese gleichzeitig zu behandeln, falls die Änderungen *sich gegenseitig beeinflussen*. Eine simple Methode wird vorgestellt, die den gegenseitigen Einfluss von Änderungen misst.

- Der verbesserte Ansatz erweist sich als nie schlechter als der Basis-Algorithmus und ist bis zu 15-mal schneller. Damit ist er der schnellste Algorithmus für die meisten betrachteten Szenarien mit sich beeinflussenden, multiplen Kantenänderungen.

Entwurf von praktikablen Verfahren für das Bahn-Verzögerungsmanagment. Bahn-Verzögerungsmanagement beschäftigt sich mit der Fragestellung wie auf exogene Verspätungen im Zugverkehr zu reagieren ist. Diese Arbeit befasst sich mit Verzögerungsmanagement für den Personenverkehr und versucht die Gesamtverspätung aller Passagiere zu minimieren.

In der wissenschaftlichen Literatur werden hauptsächlich zwei Aspekte betrachtet: Warte-Entscheidungen legen fest, ob Anschlusszüge auf verspätete Zubringer warten sollen. Vorfahrts-Entscheidungen hingegen fragen, welcher Zug bevorzugt werden soll, wenn mehrere Züge gleichzeitig auf das gleiche Stück Schieneninfrastruktur zugreifen wollen. Dies sind auch die Hauptaspekte in dieser Arbeit. Ein zusätzliches Hindernis ist der Umstand, dass exogene Verspätungen häufig nicht vorher bekannt sind. Dieser *Online-Fall* wird betrachtet. Dazu werden bestehende Modelle auf den Online-Fall erweitert und simple und praktikable Lösungsalgorithmen entwickelt. Eine Verzögerungsmanagement-Strategie bezeichnen wir im weitesten Sinne als praktikabel, wenn sie einfach und robust genug ist, um in der Praxis eingesetzt werden zu können. Eine aufwendigere Vorberechnung sei aber erlaubt.

Es stellt sich heraus, dass ein simples, simulationsbasiertes Lernverfahren Ergebnisse erzielt, die nahe am Optimum liegen. Eine interessante und hilfreiche Beobachtung ist, dass optimale Lösungen größtenteils durch eine sehr einfache und lokale Entscheidungsregel erzeugt werden können.

Curriculum Vitæ

Personal Information

Reinhard Bauer, born April 4, 1981 in Feuchtwangen, Germany

Current Status

since 03/2007	**Research and teaching assistant** Chair *Algorithmics I* (Prof. Dr. Dorothea Wagner) Karlsruhe Institute of Technology (former Universität Karlsruhe (TH))

Education

06/2000	**Abitur (German university entrance qualification)** Gymnasium Dinkelsbühl
10/2000 - 8/2001	**Alternative civilian service** Kinder- und Jugendheim Sonnenhof
10/2001 - 2/2007	**Study of mathematics with minor computer science** Universität Karlsruhe (TH)
2/2007	**Diploma in mathematics with minor computer science** Universität Karlsruhe (TH)
since 03/2007	**Research and teaching assistant** Chair *Algorithmics I* (Prof. Dr. Dorothea Wagner) Karlsruhe Institute of Technology (former Universität Karlsruhe (TH))

Journal articles

[1] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. *Networks*, 57(1):38–52, January 2011.

[2] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 15(2.3):1–31, January 2010. Special Section devoted to WEA'08.

[3] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM Journal of Experimental Algorithmics*, 14(2.4):1–29, August 2009. Special Section on Selected Papers from ALENEX 2008.

Conference articles

[1] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. Preprocessing Speed-Up Techniques is Hard. In *Proceedings of the 7th Conference on Algorithms and Complexity (CIAC'10)*, volume 6078 of *Lecture Notes in Computer Science*, pages 359–370. Springer, 2010.

[2] Reinhard Bauer, Marcus Krug, Sascha Meinert, and Dorothea Wagner. Synthetic Road Networks. In *Proceedings of the 6th International Conference on Algorithmic Aspects in Information and Management (AAIM'10)*, volume 6124 of *Lecture Notes in Computer Science*, pages 46–57. Springer, 2010.

[3] Reinhard Bauer, Marcus Krug, and Dorothea Wagner. Enumerating and Generating Labeled k-Degenerate Graphs. In *Proceedings of the Seventh Workshop on Analytic Algorithmics and Combinatorics (ANALCO '10)*, pages 90–98. SIAM, 2010.

[4] Reinhard Bauer, Gianlorenzo D'Angelo, Daniel Delling, and Dorothea Wagner. The Shortcut Problem – Complexity and Approximation. In *Proceedings of the 35th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'09)*, volume 5404 of *Lecture Notes in Computer Science*, pages 105–116. Springer, January 2009.

[5] Reinhard Bauer and Dorothea Wagner. Batch Dynamic Single-Source Shortest-Path Algorithms: An Experimental Study. In *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*, volume 5526 of *Lecture Notes in Computer Science*, pages 51–62. Springer, June 2009.

[6] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. In *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*, pages 13–26. SIAM, April 2008.

[7] Reinhard Bauer, Daniel Delling, Peter Sanders, Dennis Schieferdecker, Dominik Schultes, and Dorothea Wagner. Combining Hierarchical and Goal-Directed Speed-Up Techniques for Dijkstra's Algorithm. In *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 303–318. Springer, June 2008.

[8] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. In *Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07)*, pages 209–225. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

Bibliography

[ACG$^+$02] Giorgio Ausiello, Pierluigi Crescenzi, Giorgio Gambosi, Viggo Kann, and Alberto Marchetti-Spaccamela. *Complexity and Approximation - Combinatorial Optimization Problems and Their Approximability Properties.* Springer, 2nd edition, 2002.

[AFGW10] Ittai Abraham, Amos Fiat, Andrew V. Goldberg, and Renato F. Werneck. Highway Dimension, Shortest Paths, and Provably Efficient Algorithms. In Moses Charikar, editor, *Proceedings of the 21st Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'10)*, pages 782–793. SIAM, 2010.

[ALE06] *Proceedings of the 8th Workshop on Algorithm Engineering and Experiments (ALENEX'06).* SIAM, 2006.

[BCD$^+$08] Francesco Bruera, Serafino Cicerone, Gianlorenzo D'Angelo, Gabriele Di Stefano, and Daniele Frigioni. Dynamic Multi-level Overlay Graphs for Shortest Paths. *Mathematics in Computer Science*, 1(4):709–736, April 2008.

[BCK$^+$10a] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. Preprocessing Speed-Up Techniques is Hard. In *Proceedings of the 7th Conference on Algorithms and Complexity (CIAC'10)*, volume 6078 of *Lecture Notes in Computer Science*, pages 359–370. Springer, 2010.

[BCK$^+$10b] Reinhard Bauer, Tobias Columbus, Bastian Katz, Marcus Krug, and Dorothea Wagner. Preprocessing Speed-Up Techniques is Hard. Technical Report 2010-04, ITI Wagner, Faculty of Informatics, Karlsruhe Institute of Technology, 2010.

[BD08] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. In Ian Munro and Dorothea Wagner, editors, *Proceedings of the 10th Workshop on Algorithm Engineering and Experiments (ALENEX'08)*, pages 13–26. SIAM, April 2008.

[BD09] Reinhard Bauer and Daniel Delling. SHARC: Fast and Robust Unidirectional Routing. *ACM Journal of Experimental Algorithmics*, 14(2.4):1–29, August 2009. Special Section on Selected Papers from ALENEX 2008.

[BDD$^+$10] Reinhard Bauer, Gianlorenzo D'Angelo, Daniel Delling, Andrea Schumm, and Dorothea Wagner. The Shortcut Problem – Complexity and Algorithms . Technical Report 2010-17, ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH), 2010.

[BDDW09] Reinhard Bauer, Gianlorenzo D'Angelo, Daniel Delling, and Dorothea Wagner. The Shortcut Problem – Complexity and Approximation. In *Proceedings of the 35th International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM'09)*, volume 5404 of *Lecture Notes in Computer Science*, pages 105–116. Springer, January 2009.

[BDGW10] Edith Brunel, Daniel Delling, Andreas Gemsa, and Dorothea Wagner. Space-Efficient SHARC-Routing. In Paola Festa, editor, *Proceedings of the 9th International Symposium on Experimental Algorithms (SEA'10)*, volume 6049 of *Lecture Notes in Computer Science*, pages 47–58. Springer, May 2010.

[BDW07] Reinhard Bauer, Daniel Delling, and Dorothea Wagner. Experimental Study on Speed-Up Techniques for Timetable Information Systems. In Christian Liebchen, Ravindra K. Ahuja, and Juan A. Mesa, editors, *Proceedings of the 7th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'07)*, pages 209–225. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, 2007.

[Bel58] Richard Bellman. On a Routing Problem. *Quarterly of Applied Mathematics*, 16:87–90, 1958.

[BFM$^+$07] Holger Bast, Stefan Funke, Domagoj Matijevic, Peter Sanders, and Dominik Schultes. In Transit to Constant Shortest-Path Queries in Road Networks. In *Proceedings of the 9th Workshop on Algorithm Engineering and Experiments (ALENEX'07)*, pages 46–59. SIAM, 2007.

[BFM09] Holger Bast, Stefan Funke, and Domagoj Matijevic. Ultrafast Shortest-Path Queries via Transit Nodes. In Demetrescu et al. [DGJ09], pages 175–192.

[BHLS07] Andre Berger, Ralf Hoffmann, Ulf Lorenz, and Sebastian Stiller. Online Delay Management: PSPACE Hardness and Simulation. Technical Report 0097, ARRIVAL Project, 2007.

[BKMW10] Reinhard Bauer, Marcus Krug, Sascha Meinert, and Dorothea Wagner. Synthetic Road Networks. In *Proceedings of the 6th International Conference on Algorithmic Aspects in Information and Management (AAIM'10)*, volume 6124 of *Lecture Notes in Computer Science*, pages 46–57. Springer, 2010.

[BRT08] Luciana Buriol, Mauricio Resende, and Mikkel Thorup. Speeding Up Dynamic Shortest-Path Algorithms. *Informs Journal on Computing*, 20(2):191–204, 2008.

[BW09a] Reinhard Bauer and Dorothea Wagner. Batch Dynamic Single-Source Shortest-Path Algorithms: An Experimental Study. In Jan Vahrenhold, editor, *Proceedings of the 8th International Symposium on Experimental Algorithms (SEA'09)*, volume 5526 of *Lecture Notes in Computer Science*, pages 51–62. Springer, June 2009.

[BW09b] Reinhard Bauer and Dorothea Wagner. Batch Dynamic Single-Source Shortest-Path Algorithms: An Experimental Study. Technical Report 2009,6, ITI Wagner, Faculty of Informatics, Universität Karlsruhe (TH), 2009.

[CAI08] CAIDA: The Cooperative Association for Internet Data Analysis. `http://www.caida.org/`, 2008.

[CGR96] Boris V. Cherkassky, Andrew V. Goldberg, and Tomasz Radzik. Shortest paths algorithms. *Mathematical Programming, Series A*, 73:129–174, 1996.

[CLRS01] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms*. MIT Press, 2nd edition, 2001.

[Col09] Tobias Columbus. On the Complexity of Contraction Hierarchies, 2009. Student's thesis - Karlsruhe Institute of Technology - ITI Wagner.

[Del08] Daniel Delling. Time-Dependent SHARC-Routing. In *Proceedings of the 16th Annual European Symposium on Algorithms (ESA'08)*, volume 5193 of *Lecture Notes in Computer Science*, pages 332–343. Springer, September 2008.

[Del09] Daniel Delling. *Engineering and Augmenting Route Planning Algorithms*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2009.

[Dem01] Camil Demetrescu. *Fully Dynamic Algorithms for Path Problems on Directed Graphs*. PhD thesis, University of Rome La Sapienza, Department of Computer and Systems Science, April 2001.

[Dem07] Camil Demetrescu, editor. *Proceedings of the 6th Workshop on Experimental Algorithms (WEA'07)*, volume 4525 of *Lecture Notes in Computer Science*. Springer, June 2007.

[DGJ06] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *9th DIMACS Implementation Challenge - Shortest Paths*, November 2006.

[DGJ09] Camil Demetrescu, Andrew V. Goldberg, and David S. Johnson, editors. *The Shortest Path Problem: Ninth DIMACS Implementation Challenge*, volume 74 of *DIMACS Book*. American Mathematical Society, 2009.

[DHM+09] Daniel Delling, Martin Holzer, Kirill Müller, Frank Schulz, and Dorothea Wagner. High-Performance Multi-Level Routing. In Demetrescu et al. [DGJ09], pages 73–92.

[DHSS09] Twan Dollevoet, Dennis Huisman, Marie Schmidt, and Anita Schöbel. Delay Management with Re-Routing of Passengers. In *Proceedings of the 9th Workshop on Algorithmic Approaches for Transportation Modeling, Optimization, and Systems (ATMOS'09)*, Dagstuhl Seminar Proceedings, pages 1–17, 2009.

[DI06] Camil Demetrescu and Giuseppe F. Italiano. Dynamic shortest paths and transitive closure: Algorithmic techniques and data structures. *Journal of Discrete Algorithms*, 4(3):353–383, September 2006.

[Dij59] Edsger W. Dijkstra. A Note on Two Problems in Connexion with Graphs. *Numerische Mathematik*, 1:269–271, 1959.

[DSSW09] Daniel Delling, Peter Sanders, Dominik Schultes, and Dorothea Wagner. Engineering Route Planning Algorithms. In Jürgen Lerner, Dorothea Wagner, and Katharina A. Zweig, editors, *Algorithmics of Large and Complex Networks*, volume 5515 of *Lecture Notes in Computer Science*, pages 117–139. Springer, 2009.

[DW07] Daniel Delling and Dorothea Wagner. Landmark-Based Routing in Dynamic Graphs. In Demetrescu [Dem07], pages 52–65.

[EG08] David Eppstein and Michael T. Goodrich. Studying (non-planar) road networks through an algorithmic lens. In *Proceedings of the 16th ACM SIGSPATIAL international conference on Advances in geographic information systems (GIS '08)*, pages 1–10. ACM Press, 2008.

[FINP98] Daniele Frigioni, Mario Ioffreda, Umberto Nanni, and Giulio Pasqualone. Experimental Analysis of Dynamic Algorithms for the Single Source Shortest Path Problem. *ACM Journal of Experimental Algorithmics*, 3(5):1–20, 1998.

[FMSN00] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully Dynamic Algorithms for Maintaining Shortest Paths trees. *Journal of Algorithms*, 34(2):251–281, February 2000.

[FMSN03] Daniele Frigioni, Alberto Marchetti-Spaccamela, and Umberto Nanni. Fully dynamic shortest paths in digraphs with arbitrary arc weights. *Journal of Algorithms*, 49(1):86–113, October 2003.

[Fuc10] Fabian Fuchs. On Preprocessing the ALT-Algorithm, 2010. Student's thesis - Karlsruhe Institute of Technology - ITI Wagner.

[Gat07] Michael Gatto. *On the impact of uncertainty on some optimization problems* . PhD thesis, ETH Zürich, 2007.

[GGP+04] Michael Gatto, Björn Glaus, Leon Peeters, Riko Jacob, and Peter Widmayer. Railway Delay Management: Exploring Its Algorithmic Complexity . In *Proceedings of the 9th Scandinavian Workshop on Algorithm Theory (SWAT'04)*, volume 3111 of *Lecture Notes in Computer Science*, pages 199–211. Springer, 2004.

[GH05] Andrew V. Goldberg and Chris Harrelson. Computing the Shortest Path: A* Search Meets Graph Theory. In *Proceedings of the 16th Annual ACM–SIAM Symposium on Discrete Algorithms (SODA'05)*, pages 156–165. SIAM, 2005.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability. A Guide to the Theory of $\mathcal{NP}$-Completeness.* W. H. Freeman and Company, 1979.

[GJPS05] Michael Gatto, Riko Jacob, Leon Peeters, and Anita Schöbel. The computational complexity of delay management. In *Proceedings of the 31th International Workshop on Graph-Theoretic Concepts in Computer Science (WG'05)*, volume 3787 of *Lecture Notes in Computer Science*, pages 227–238. Springer, 2005.

[GJPW07] Michael Gatto, Riko Jacob, Leon Peeters, and Peter Widmayer. On-line delay management on a single train line. In *Algorithmic Methods for Railway Optimization* [GKS$^+$07].

[GKS$^+$07] Frank Geraets, Leo G. Kroon, Anita Schöbel, Dorothea Wagner, and Christos Zaroliagis. *Algorithmic Methods for Railway Optimization*, volume 4359 of *Lecture Notes in Computer Science*. Springer, 2007.

[GKW06] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Efficient Point-to-Point Shortest Path Algorithms. In ALENEX'06 [ALE06], pages 129–143.

[GKW07] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Better Landmarks Within Reach. In Demetrescu [Dem07], pages 38–51.

[GKW09] Andrew V. Goldberg, Haim Kaplan, and Renato F. Werneck. Reach for A*: Shortest Path Algorithms with Preprocessing. In Demetrescu et al. [DGJ09], pages 93–139.

[Gov98] Rob Goverde. The Max-Plus Algebra Approach to Railway Timetable Design . In *Computers in Railways VI*, pages 339–350. WIT Press, 1998.

[GS07] Andreas Ginkel and Anita Schöbel. To wait or not to wait? The bicriteria delay management problem in public transportation. *Transportation Science*, 41(4):527–538, 2007.

[GSSD08] Robert Geisberger, Peter Sanders, Dominik Schultes, and Daniel Delling. Contraction Hierarchies: Faster and Simpler Hierarchical Routing in Road Networks. In Catherine C. McGeoch, editor, *Proceedings of the 7th Workshop on Experimental Algorithms (WEA'08)*, volume 5038 of *Lecture Notes in Computer Science*, pages 319–333. Springer, June 2008.

[Gut04] Ronald J. Gutman. Reach-Based Routing: A New Approach to Shortest Path Algorithms Optimized for Road Networks. In *Proceedings of the 6th Workshop on Algorithm Engineering and Experiments (ALENEX'04)*, pages 100–111. SIAM, 2004.

[GW05] Andrew V. Goldberg and Renato F. Werneck. Computing Point-to-Point Shortest Paths from External Memory. In *Proceedings of the 7th Workshop on Algorithm Engineering and Experiments (ALENEX'05)*, pages 26–40. SIAM, 2005.

[HaC08] HaCon - Ingenieurgesellschaft mbH. `http://www.hacon.de`, 2008.

[HGL08] Geraldine Heilporn, Luigi De Giovanni, and Martine Labbe. Optimization models for the single delay management problem in public transportation. *European Journal of Operational Research*, 189:762–774, 2008.

[HKMS06] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Demetrescu et al. [DGJ06].

[HKMS09] Moritz Hilger, Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Fast Point-to-Point Shortest Path Computations with Arc-Flags. In Demetrescu et al. [DGJ09], pages 41–72.

[HMU07] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation.* Pearson International Edition, 2007.

[HNR68] Peter E. Hart, Nils Nilsson, and Bertram Raphael. A Formal Basis for the Heuristic Determination of Minimum Cost Paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.

[Hoe63] Wassily Hoeffding. Probability inequalities for sums of bounded random variables. *Journal of the American Statistical Association*, 58(301):713–721, 1963.

[Hol08] Martin Holzer. *Engineering Planar-Separator and Shortest-Path Algorithms.* PhD thesis, Karlsruhe Institute of Technology (KIT) - Department of Informatics, 2008.

[HSW06] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. In ALENEX'06 [ALE06], pages 156–170.

[HSW08] Martin Holzer, Frank Schulz, and Dorothea Wagner. Engineering Multi-Level Overlay Graphs for Shortest-Path Queries. *ACM Journal of Experimental Algorithmics*, 13(2.5):1–26, December 2008.

[Jun99] Dieter Jungnickel. *Graphs, Networks and Algorithms*, volume 5 of *Algorithms and Computation in Mathmatics.* Springer, 1999.

[KMS05] Ekkehard Köhler, Rolf H. Möhring, and Heiko Schilling. Acceleration of Shortest Path and Constrained Shortest Path Computation. In WEA'05 [WEA05], pages 126–138.

[KS10] Natalia Kliewer and Leena Suhl. A Note on the Online Nature of the Railway Delay Management Problem. *Networks*, 2010.

[KT01] Valerie King and Mikkel Thorup. A space saving trick for directed dynamic transitive closure and shortest path algorithms. In *Proceedings of the 7th Annual International Conference on Computing Combinatorics (COCOON'01)*, volume 2108 of *Lecture Notes in Computer Science*, pages 268–277. Springer, 2001.

[Lau04] Ulrich Lauther. An Extremely Fast, Exact Algorithm for Finding Shortest Paths in Static Networks with Geographical Background. In *Geoinformation und Mobilität - von der Forschung zur praktischen Anwendung*, volume 22, pages 219–230. IfGI prints, 2004.

[MSS$^+$05] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speed Up Dijkstra's Algorithm. In WEA'05 [WEA05], pages 189–202.

[MSS$^+$06] Rolf H. Möhring, Heiko Schilling, Birk Schütz, Dorothea Wagner, and Thomas Willhalm. Partitioning Graphs to Speedup Dijkstra's Algorithm. *ACM Journal of Experimental Algorithmics*, 11(2.8):1–29, 2006.

[Mül06] Kirill Müller. Design and Implementation of an Efficient Hierarchical Speed-up Technique for Computation of Exact Shortest Paths in Graphs. Master's thesis, Universität Karlsruhe (TH), Fakultät für Informatik, June 2006.

[NST00] Paolo Narváez, Kai-Yeung Siu, and Hong-Yi Tzeng. New Dynamic Algorithms for Shortest Path Tree Computation. *IEEE/ACM Transactions on Networking*, 8(6):734–746, 2000.

[NW88] Georg L. Nemhauser and Laurence A. Wolsey. *Integer and Combinatorial Optimization.* Wiley, 1988.

[Pap74] U. Pape. Implementation and Efficiency of Moore-Algorithms for the Shortest Route Problem. *Mathematical Programming*, 7:212–222, 1974.

[PTV08] PTV AG - Planung Transport Verkehr. `http://www.ptv.de`, 2008.

[RR96a] G. Ramalingam and Thomas Reps. On the computational complexity of dynamic graph problems. *Theoretical Computer Science*, 158(1):233–277, May 1996.

[RR96b] Thomas Reps and G. Ramalingam. An Incremental Algorithm for a Generalization of the Shortest-Path Problem. *Journal of Algorithms*, 21(2):267–305, September 1996.

[Sch01] Anita Schöbel. A model for the delay management problem based on mixed-integer programming. *Electronic Notes in Theoretical Computer Science*, 50(1):1–10, 2001.

[Sch05] Frank Schulz. *Timetable Information and Shortest Paths.* PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2005.

[Sch06a] Heiko Schilling. *Route Assignment Problems in Large Networks.* PhD thesis, Technische Universität Berlin, 2006.

[Sch06b] Anita Schöbel. *Optimization in Public Transportation*, volume 3 of *Springer Optimization and Its Applications.* Springer, 2006.

[Sch07] Anita Schöbel. Integer programming approaches for solving the delay management problem. In *Algorithmic Methods for Railway Optimization* [GKS$^+$07], pages 145–170.

[Sch08] Dominik Schultes. *Route Planning in Road Networks.* PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, February 2008. `http://algo2.iti.uka.de/schultes/hwy/schultes_diss.pdf`.

[Sch09a] Anita Schöbel. Capacity constraints in delay management. *Public Transport*, 1(2):135–154, 2009.

[Sch09b] Andrea Schumm. Heuristic Algorithms for the Shortcut Problem. Master's thesis, Karlsruhe Institute of Technology (KIT), July 2009.

[Sch10] Michael Schachtebeck. *Delay Management in Public Transportation: Capacities, Robustness, and Integration.* PhD thesis, Georg-August-Universität Göttingen, 2010.

[Slo08] Neil James Alexander Sloane. The On-Line Encyclopedia of Integer Sequences, 2008. `www.research.att.com/~njas/sequences/`.

[SMB01] Leena Suhl, Taieb Mellouli, and Claus Biederbick. Managing and preventing delays in railway traffic. In Matti Pursula and Jarko Niittymäki, editors, *Mathematical methods on Optimization in Transportation Systems*, pages 3–16. 2001.

[SS05] Peter Sanders and Dominik Schultes. Highway Hierarchies Hasten Exact Shortest Path Queries. In *Proceedings of the 13th Annual European Symposium on Algorithms (ESA'05)*, volume 3669 of *Lecture Notes in Computer Science*, pages 568–579. Springer, 2005.

[SS06a] Peter Sanders and Dominik Schultes. Engineering Highway Hierarchies. In *Proceedings of the 14th Annual European Symposium on Algorithms (ESA'06)*, volume 4168 of *Lecture Notes in Computer Science*, pages 804–816. Springer, 2006.

[SS06b] Peter Sanders and Dominik Schultes. Robust, Almost Constant Time Shortest-Path Queries in Road Networks. In Demetrescu et al. [DGJ06].

[SS07] Dominik Schultes and Peter Sanders. Dynamic Highway-Node Routing. In Demetrescu [Dem07], pages 66–79.

[SS09] Michael Schachtebeck and Anita Schöbel. LinTim - A Toolbox for the Experimental Evaluation of the Interaction of Different Planning Stages in Public Transportation. Technical Report 0206, ARRIVAL Project, 2009.

[SS10] Michael Schachtebeck and Anita Schöbel. To wait or not to wait and who goes first? Delay Management with Priority Decisions. *Transportation Science*, 44(3):307–321, 2010.

[SWW99] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. In *Proceedings of the 3rd International Workshop on Algorithm Engineering (WAE'99)*, volume 1668 of *Lecture Notes in Computer Science*, pages 110–123. Springer, 1999.

[SWW00] Frank Schulz, Dorothea Wagner, and Karsten Weihe. Dijkstra's Algorithm On-Line: An Empirical Case Study from Public Railroad Transport. *ACM Journal of Experimental Algorithmics*, 5(12):1–23, 2000.

[SWZ02] Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. Using Multi-Level Graphs for Timetable Information in Railway Systems. In *Proceedings of the 4th Workshop on Algorithm Engineering and Experiments (ALENEX'02)*, volume 2409 of *Lecture Notes in Computer Science*, pages 43–59. Springer, 2002.

[TTIW07] Satoshi Taoka, Daisuke Takafuji, Takashi Iguchi, and Toshimasa Watanabe. Performance Comparison of Algorithms for the Dynamic Shortest Path Problem. *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, E90-A(4), April 2007.

[Uni08] University of Oregon Routeviews Project. `http://www.routeviews.org/`, 2008.

[VSM98] Remco De Vries, Bart De Schutter, and Bart De Moor. On max-algebraic models for transportation networks. In *Proceedings of the 4th International Workshop on Discrete Event Systems (WODES '98)*, pages 457–462, 1998.

[WEA05] *Proceedings of the 4th Workshop on Experimental Algorithms (WEA'05)*, volume 3503 of *Lecture Notes in Computer Science*. Springer, 2005.

[Wil05] Thomas Willhalm. *Engineering Shortest Paths and Layout Algorithms for Large Graphs*. PhD thesis, Universität Karlsruhe (TH), Fakultät für Informatik, 2005.

[WW03] Dorothea Wagner and Thomas Willhalm. Geometric Speed-Up Techniques for Finding Shortest Paths in Large Sparse Graphs. In *Proceedings of the 11th Annual European Symposium on Algorithms (ESA'03)*, volume 2832 of *Lecture Notes in Computer Science*, pages 776–787. Springer, 2003.

[WW07a] Dorothea Wagner and Roger Wattenhofer, editors. *Algorithms for Sensor and Ad Hoc Networks*, volume 4621 of *Lecture Notes in Computer Science*. Springer, 2007.

[WW07b] Dorothea Wagner and Thomas Willhalm. Speed-Up Techniques for Shortest-Path Computations. In *Proceedings of the 24th International Symposium on Theoretical Aspects of Computer Science (STACS'07)*, volume 4393 of *Lecture Notes in Computer Science*, pages 23–36. Springer, 2007.

[WWZ05] Dorothea Wagner, Thomas Willhalm, and Christos Zaroliagis. Geometric Containers for Efficient Shortest-Path Computation. *ACM Journal of Experimental Algorithmics*, 10(1.3):1–30, 2005.