The Shortest Path Problem
on Real Road Networks
Theory, Algorithms and Computations

Proefschrift

ter verkrijging van de graad van doctor
aan de Technische Universiteit Delft,
op gezag van de Rector Magnificus prof. ir. K. Ch. A. M. Luyben,
voorzitter van het College voor Promoties,
in het openbaar te verdedigen op woensdag 26 februari 2014 om 12:30 uur
door Hendrik Nicolaas POST,
wiskundig ingenieur,
geboren te Delft.
Dit proefschrift is goedgekeurd door de promotor:
Prof. dr. ir. K.I. Aardal

Samenstelling promotiecommissie:

Rector Magnificus voorzitter
Prof. dr. ir. K.I. Aardal Technische Universiteit Delft, promotor
Prof. dr. ir. C. Roos Technische Universiteit Delft
Prof. dr. ir. D. den Hertog Universiteit van Tilburg
Prof. dr. R.H. Möhring Technische Universität Berlin
Prof. dr. C. Witteveen Technische Universiteit Delft
Prof. dr. ing. G.J. Woeginger Technische Universiteit Eindhoven
Prof. dr. T. van Woensel Technische Universiteit Eindhoven

Prof. dr. ir. C. Roos heeft als begeleider in belangrijke mate aan de totstandkoming van het proefschrift bijgedragen.

Het onderzoek in dit proefschrift is mede gefinancierd door Connexxion.

Omslag:
Henk de Roos, *Porsche 011 Carrera, 1963* (detail), viltstift op papier, 70 x 100 cm. Kunstuitleencollectie Special Arts.


Copyright © 2014 by H.N. Post

All rights reserved. No part of the material protected by this copyright notice may be reproduced or utilized in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage and retrieval system, without written permission of the author.

Printed in The Netherlands by Real Concepts B.V., Duiven.
Acknowledgements

After completing my Masters thesis in 1995, supervised by Kees Roos, I started working on real-time vehicle routing problems. About two years later, I invited Kees to show him some results of my work. He then advised me to write a PhD thesis on the topic. Although the idea immediately appealed to me, I only formally started my PhD research in 2006. Due to a full-time job, I had to pause the project on several occasions. I thank Kees, not only for always motivating me to keep on studying, but also for being patient and having a continuous belief that I would actually one day finish the project.

From 2010, I worked on this PhD on a more regular basis. By that time, Kees reached the age where he lost his ‘ius promovendi’. He therefore asked Karen Aardal to be my new supervisor. I thank Karen for accepting this. Furthermore, I am very thankful that she asked Kees to continue his role as supervisor. Karen proposed to have monthly sessions to keep the pace. Although I was wondering whether or not I would be able to have some new material to discuss every month, I now realise that my PhD is only finished due to those monthly sessions. Karen, I am very thankful to you for your involvement. We will meet each other in the REPRO project and I hope we can continue our cooperation in several other projects as well.

Especially during the past four years, actually writing this thesis required a lot of patience from my colleagues and family. I will therefore thank my closest colleagues: Age van den Berg, Arthur Bosma, Bert Quak, Erik Zijp, Harte Oldenziel and Hyme Karels. Every day I am happy to be at the office. I appreciate the open atmosphere and the way we work together. Over the years, I have worked with many other colleagues, but this list of people is too long to mention here. But I have to mention Rudy Selles and Michiel Bijleveld. You both accepted that I needed time to do this research. Thanks are also due to Bart Willemsen, who read my thesis three times over and corrected my English where necessary. Pim Lingsma, thanks for helping me with the cover.

Finally, I thank my family. My wife Maartje and our children Rik, Ruben and Rosalyn. I am grateful for your patience, especially during the final years of this PhD project. Every day, you show me interesting things in life, other than mathematics and computer programming. I am in your debt.

Above all, I thank You, Lord, for the strength and ability to complete this project.

Nijkerk, January 2014

Henk Post
Acknowledgements v

1 Introduction 3
  1.1 A brief history ................................................. 3
  1.2 Our contributions .............................................. 5
  1.3 Outline of this thesis .......................................... 7

2 Mathematical description 9
  2.1 Algorithms and running times .................................. 9
    2.1.1 Alphabet ................................................. 9
    2.1.2 Turing machine ......................................... 10
    2.1.3 Algorithms ............................................. 11
    2.1.4 Random access machine ................................ 11
    2.1.5 Asymptotic analysis .................................... 13
    2.1.6 Parameter balancing .................................... 14
    2.1.7 Amortized analysis ..................................... 14
  2.2 Notation and definitions ...................................... 16
    2.2.1 Graphs .................................................. 16
    2.2.2 Paths and cycles ....................................... 17
    2.2.3 Connectivity and components .......................... 18
    2.2.4 Forward and backward star ............................ 19
    2.2.5 Arc cost and path length ............................. 19
    2.2.6 Shortest Path Tree ................................... 20
2.2.7 Assumptions .............................................. 21
2.3 Formulation as a Minimum Cost Flow Problem .............. 21
  2.3.1 Formulation of the SSSD-SPP .......................... 22
  2.3.2 Formulation of the MSMD-SPP ......................... 23
2.4 The dual problem ........................................... 23
  2.4.1 Optimality conditions for the SPP ..................... 24
  2.4.2 Complementary slackness conditions for the SPP ........ 25
  2.4.3 Bellman’s Equations ................................. 26

3 Different methods for solving the SPP .......................... 27
  3.1 Generic Shortest Path algorithm .......................... 27
    3.1.1 Keeping track of the shortest path tree .............. 27
    3.1.2 Two common subroutines ............................... 28
    3.1.3 Generic Shortest Path algorithm ..................... 28
    3.1.4 A special case: unit lengths .......................... 31
  3.2 Label-setting algorithms .................................. 32
    3.2.1 Improved stop condition .............................. 33
    3.2.2 Complexity ........................................... 33
    3.2.3 Data structures ....................................... 34
  3.3 Label-correcting algorithms ............................... 36
    3.3.1 Potential-invariant algorithms ....................... 36
    3.3.2 Advanced label-correcting algorithms ............... 45
    3.3.3 Some optimization strategies ........................ 52
    3.3.4 Stop condition for label-correcting algorithms .... 54
  3.4 Arc set partition .......................................... 55

4 Bidirectional search ........................................ 57
  4.1 Notation .................................................. 58
  4.2 Bidirectional label-setting ............................... 58
    4.2.1 Stop condition ...................................... 58
    4.2.2 Forward or backward? ................................. 62
  4.3 Bidirectional label-correcting ............................ 63
## 5 Heuristic estimators

5.1 The family of $A^*$ algorithms ................................. 65

5.1.1 Consistent heuristic estimators .......................... 67
5.1.2 The star in $A^*$ ............................................ 68
5.1.3 Commonly used consistent heuristic estimators .............. 71
5.1.4 Equivalency between DIJKSTRA and $A^*$ ................. 73

5.2 Label-correcting algorithms and heuristic estimators .......... 74

5.2.1 Adding heuristic estimators to label-correcting algorithms ... 75
5.2.2 Inconsistent heuristic estimators ........................... 76
5.2.3 Variable heuristic estimates ............................... 77

5.3 Bidirectional algorithms and heuristic estimators .............. 79

5.3.1 The symmetric approach .................................. 79
5.3.2 The balanced approach .................................... 81
5.3.3 A new symmetric approach ............................... 83
5.3.4 Divide and conquer ....................................... 87

## 6 Preprocessing the graph

6.1 Test environment .............................................. 93

6.1.1 Road map of TomTom .................................... 93
6.1.2 Hardware and compiler ................................... 94

6.2 Static graph preprocessing .................................... 95

6.2.1 Quarters .................................................. 95
6.2.2 Bypass degree 2 nodes ................................... 96
6.2.3 Partitioning the node set $V$ ............................... 99
6.2.4 Summary .................................................. 99

6.3 Dynamic graph preprocessing .................................. 101

6.3.1 Graph contraction ........................................ 102
6.3.2 Reaches .................................................. 105
6.3.3 Landmarks ............................................... 108

6.4 Contraction Hierarchies ....................................... 116

6.4.1 Construction of Contraction Hierarchies .................. 116
### Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.4.2 Solving the SSSD-SPP with Contraction Hierarchies</td>
<td>119</td>
</tr>
<tr>
<td>6.4.3 Determining a node ordering</td>
<td>120</td>
</tr>
<tr>
<td>6.4.4 Hub Based Labeling</td>
<td>122</td>
</tr>
<tr>
<td>6.5 Hybrid graph preprocessing</td>
<td>130</td>
</tr>
<tr>
<td>6.5.1 The graph partitioning problem</td>
<td>131</td>
</tr>
<tr>
<td>6.5.2 Arc flags</td>
<td>145</td>
</tr>
<tr>
<td>7 Computational results</td>
<td>151</td>
</tr>
<tr>
<td>7.1 Experimental setup</td>
<td>151</td>
</tr>
<tr>
<td>7.1.1 Comparison of optimization algorithms</td>
<td>152</td>
</tr>
<tr>
<td>7.1.2 Estimators that are compared</td>
<td>153</td>
</tr>
<tr>
<td>7.1.3 Bidirectional methods</td>
<td>154</td>
</tr>
<tr>
<td>7.1.4 Preprocessing techniques</td>
<td>155</td>
</tr>
<tr>
<td>7.1.5 Test set</td>
<td>155</td>
</tr>
<tr>
<td>7.2 Experiments on the full graph</td>
<td>156</td>
</tr>
<tr>
<td>7.2.1 Unidirectional search on the full graph</td>
<td>156</td>
</tr>
<tr>
<td>7.2.2 Bidirectional search on the full graph</td>
<td>163</td>
</tr>
<tr>
<td>7.2.3 Partial shortest path trees</td>
<td>172</td>
</tr>
<tr>
<td>7.3 Statically preprocessed graph</td>
<td>174</td>
</tr>
<tr>
<td>7.4 Dynamically preprocessed graph</td>
<td>181</td>
</tr>
<tr>
<td>7.5 Hybrid preprocessing of graphs</td>
<td>188</td>
</tr>
<tr>
<td>7.6 Contraction Hierarchies</td>
<td>195</td>
</tr>
<tr>
<td>7.7 Concluding remarks</td>
<td>197</td>
</tr>
<tr>
<td>A Abbreviations</td>
<td>199</td>
</tr>
<tr>
<td>B Priority queue</td>
<td>201</td>
</tr>
<tr>
<td>B.1 Heaps</td>
<td>202</td>
</tr>
<tr>
<td>B.1.1 d-Heap</td>
<td>202</td>
</tr>
<tr>
<td>B.1.2 Binomial heaps</td>
<td>206</td>
</tr>
<tr>
<td>B.1.3 Fibonacci heap</td>
<td>210</td>
</tr>
<tr>
<td>B.1.4 Relaxed heap</td>
<td>217</td>
</tr>
</tbody>
</table>
Introduction

The computation of a shortest path between different locations on a graph appears to be a key problem in many applications. The wide range of applications, as well as the simplicity of the problem, makes the Shortest Path Problem (SPP) very attractive for researchers in several disciplines.

Although the SPP is relatively easy to solve, the design and analysis of most efficient algorithms for solving the SPP require creativity and cleverness. The SPP has inspired many researchers and practitioners for decades. And one might expect that the interest in the SPP will never disappear.

1.1 A brief history

In 1735, Euler [66] wondered whether or not it was possible to find a path through Königsberg, crossing each of its seven bridges exactly once and ultimately return to the original starting point. He laid the foundation of what is now known as graph theory, which in turn paved the way for path finding algorithms.

In the early 1950s, several methods were invented that could be considered initial versions of a shortest path algorithm. Pollack and Wiebenson [160] reviewed some of these solution methods for the SPP. They mention the algorithm of Dantzig, based on the simplex method, as the first solution for the SPP that appeared in the literature [39]. They also describe a far more efficient algorithm (credited to Minty) that nowadays can be considered as a label setting algorithm. This algorithm however, most likely has to be credited to Ford and Fulkerson [71], who developed it for a more general problem concerning maximal flows in a network, of which the shortest path is a special case. A better label setting algorithm was proposed by Dijkstra [59] and independently by Dantzig [40] and Whiting and Hillier [198].

During the second half of the 1960s, new algorithms were developed that improved the existing algorithms in terms of computational efficiency. For instance, Nicholson
4 1 Introduction

[140] introduced a new method to find the shortest path between two points in a network. A shortest path is found by investigating a selection of paths both from the origin, as well as from the destination. The selection of paths is decided dynamically by extending, one by one, the paths that have covered the least distance at that moment. This method can be seen as the first bidirectional method.

In 1968, Hart [103, 104] proved that a heuristic-based approach works optimally if a so-called consistent heuristic is used. This heuristic-based approach was invented by Nilsson and later improved by Raphael. They called their algorithm $A^*$, which is still a famous algorithm today, especially in the field of Artificial Intelligence [163, 169].

Pohl [158] combined bidirectional search and heuristic-based search. During the 1970s, many new techniques and algorithms were developed for several variations of the SPP.

During the 1980s and 1990s, a lot of work has been done on improving the existing algorithms, e.g. by introducing several new data structures. For example, in [4], the radix-heap implementation was developed. In 1999, Thorup [189] proved that there is a deterministic linear time and linear space algorithm for the single-source shortest path problem regarding undirected graphs with positive integer weights. His analysis was based on so-called atomic heaps that are not directly implementable on modern computers due to the huge word length involved [5]. Other researchers used the flexibility of generic label correcting algorithms to design techniques that are very efficient in practice. Gallo and Pallottino [75] introduced an implementation that retains the computational efficiency of an earlier algorithm (of D’Esopo and Pape), which runs in polynomial time. Other researchers published algorithms for which it was uncertain whether or not they run in polynomial time, but their algorithms outperformed all or most of the others known in those days. For example, Bertsekas invented the so-called Small Label First method [18].
1.2 Our contributions

Over the past ten years, the research on the SPP has mainly focused on searching the shortest path from one origin to one destination in large scale road maps. The introduction of car navigation systems has lead to many new techniques. Most of the research concentrated on preprocessing the network first. Ertl [65] introduced such a technique in 1998. During the preprocessing, some additional information is determined. This information can be used in any SPP-algorithm to reduce the search space. In 2004, Goldberg and Harrelson [89] presented a fast algorithm at the cost of huge memory requirements. By calculating the shortest paths from all nodes to a certain set of nodes, called landmarks, they developed an estimator that could be used in an $A^*$-algorithm. In that same year, Gutman [99] described an algorithm that determines a so-called reach (see Section 6.3.2) for each node. As had been the case in Ertl’s algorithm, these reaches were used to prune the search space for any SPP algorithm. In 2005, the latter two techniques were combined by Goldberg et al. [90]. They improved the preprocessing time needed to calculate the reach of each node drastically, by introducing the notion of canonical shortest paths and adding shortcuts in the graph.

In 2005, Sanders and Schultes [170] exploited the hierarchy inherent to road maps. Their algorithm, called ‘Highway Hierarchies’, is approximately 2,000 times faster than Dijkstra’s algorithm. The idea of utilizing the hierarchy inherent to road maps was already used in 1992 by Shapiro [178], but then it lead to suboptimal solutions. Sanders and Schultes introduced a preprocessing technique such that the optimality is ensured. The concept of ‘Highway Hierarchies’ has since then lead to considerably faster algorithms, such as ‘Highway Node Routing’ [176] and ‘Transit Node routing’ [9]. In [13], several recent algorithms are compared and combined.

In 2008, Geisberger [79] assigned a unique level to each node during a preprocessing step, called Contraction Hierarchies. It is ensured that there is always a shortest path from an origin to a destination such that the path visits nodes of strictly increasing hierarchy levels up to a certain node. That node has the highest hierarchy level among all nodes on the path. From that node on, the path continues via nodes that have a lower level than each prior node. Although originally based on the work of Sanders and Schultes, this idea closely resembles the work of Chleq [31] already done in 1995. Abraham et al. [1, 2] use the idea of Contraction Hierarchies in their work to obtain a so-called labeling. From this labeling, a shortest path between any pair of nodes can be determined by just inspecting the labels of those two nodes. Currently, this method is the fastest algorithm known for solving the shortest path algorithm (see Section 6.4.4). The drawback of this algorithm is that it is not suited for updating the arc costs dynamically and it uses a huge amount of memory.

1.2 Our contributions

The SPP is well-known. While interesting in itself, it occurs in many other applications as a subproblem as well. Not surprisingly, there is a vast amount of literature on this subject. Generally speaking, there are two main leads in the literature. One
of them focusses on the development of new algorithms and efficient data structures especially for the label-setting algorithms. The other one started more or less at the same time with the development of so-called guided search (A∗) and bidirectional search. At several occasions these main leads intersect. Especially since the work of Ikeda et al. [112] in 1994, bidirectional guided search became popular. Furthermore, the landmarks of Goldberg and Harrelson [89] strongly improved the quality of the lower bounds that are used in guided search. Nowadays, the research on the SPP mainly focusses on preprocessing the graph, in order to reduce the number of searched nodes, by initially adding some data to the graph.

Our contributions are:

1. In this thesis we unify and extend literature on the SPP. Although it is impossible to incorporate all existing literature, we tried to cover the main topics in detail. We extend the literature where we encountered some loose ends. For example, bidirectional search seemed to be used only within a label-setting environment so far. In Chapter 4 we provide the theory and an algorithm (Algorithm 4.5) that makes it possible to use bidirectional search also in a label-correcting environment. This chapter is partly based on our paper [121].

2. In guided (A∗) search a heuristic estimate is used that provides a lower bound on the shortest distance between a node and a specific destination node. If guided search is used in a bidirectional environment, it was already observed by Pohl [158] and Kwa [128] that determining the right stopping criterium was difficult. Therefore most researchers follow Ikeda et al. [112] by using a so-called balanced approach. If \( V \) is the set of nodes in the graph, the forward search (from origin \( s \) to destination \( t \)) uses a heuristic estimator function \( h^f: V \to \mathbb{R} \) and the backward search (from destination \( t \) to origin \( s \), in the reverse graph) uses a heuristic estimate \( h^b: V \to \mathbb{R} \). In a balanced approach, we have \( h^f(v) + h^b(v) = 0 \), for all \( v \in V \). If \( d(v, w) \) denotes the shortest path distance from \( v \) to \( w \) (for all \( v, w \in V \)), and \( \delta_v^s \leq d(s, v) \) and \( \delta_v^t \leq d(v, t) \) for all \( v \in V \), a balanced heuristic estimates can be derived as follows:

\[
\begin{align*}
    h^f_v & \leftarrow \frac{1}{2} \delta_v^t - \frac{1}{2} \delta_v^s, \\
    h^b_v & \leftarrow \frac{1}{2} \delta_v^s - \frac{1}{2} \delta_v^t.
\end{align*}
\]

Hence, instead of using the full power of \( \delta_v^s \) in the forward search and that of \( \delta_v^t \) in the backward search, a somewhat awkward estimate is derived that uses both lower bounds \( \delta_v^s \) and \( \delta_v^t \) in each direction. The main reason for using a balanced approach, is that it can be incorporated in a plain (no estimates at all) bidirectional version of Dijkstra’s algorithm. For a bidirectional version of Dijkstra it is known that as soon as the forward and backward search meet each other at a node \( u \), only a relatively short postprocessing has to be executed to determine a shortest path. In a symmetric approach the estimate \( h^f_v \leftarrow \delta_v^s \) is used in the forward search and \( h^b_v \leftarrow \delta_v^s \) in the backward search. Until now, it is believed that a symmetric approach has the drawback that the major part
of the execution time comes after the forward and backward search intersect. We provide a new bound that can be used to reduce the time needed for this so-called post-phase. We also show that by using this bound together with balanced heuristic estimators leads to the incorporation of the well-known post-phase of the balanced algorithm directly into the algorithm itself. In this way, our new algorithm is a generalization of the balanced approach. In Chapter 5 we discuss the research on guided search in detail. We incorporate our papers [154, 155] at the end of that chapter.

3. We encountered a paper of Whangbo [197] where a different strategy is used in case there are Euclidean coordinates known for each node. Based on scalar projections, the search is divided over forward and backward iterations, after both search spaces intersect. However, we found some errors in his theory. Therefore, we provide a corrected version of this theory (see Section 5.3.4).

4. Finally, we present an experimental evaluation of 28 optimization algorithms on several (preprocessed) graphs and using different heuristics for providing a lower bound. Based on this evaluation several suggestions are made for specific situations where the SPP has to be solved. During our experiments we were able to improve the use of a landmark based heuristic estimator when used in a unidirectional way. We provide some insight into why landmark based estimators are quite useless in a symmetric approach and profits from a balance approach. Furthermore, we empirically demonstrate that after the so-called hybrid preprocessing, unidirectional methods outperform their bidirectional counterparts.

1.3 Outline of this thesis

In Chapter 2, we briefly describe the mathematical background of the SPP. Subsequently in Chapter 3, several methods to solve the SPP are described: the label-setting and label-correcting algorithms are discussed in detail. The methods in this chapter are general, in the sense that they are applicable to both single destination and multiple destinations.

In car navigation systems, like in many other situations, the shortest path from one origin to one destination has to be calculated. This so-called single-source, single-destination shortest path problem SSSD-SPP has been widely studied over the past decades. In Chapter 4, bidirectional algorithms for the SSSD-SPP are discussed. Here, we provide some new theory to make bidirectional implementations of label correcting algorithms possible [121]. For the SSSD-SPP heuristic estimates are highly effective. In Chapter 5 the theory of heuristic estimates is discussed. So far, most research on bidirectional search is based on so-called balanced heuristics. We provide some new theory in favor of symmetric heuristics [154, 155] and we correct the theory on scalar projections.
Research on SPP is not limited to the development of new optimization algorithms. Over the past ten years or so, much research has been done on preprocessing techniques. The idea is that, on a certain graph, several SPP’s have to be solved. It might therefore be a good idea to preprocess the graph in order to get information that can be used to reduce the time needed to solve the SPP’s. Several preprocessing techniques are discussed in Chapter 6.

An experimental evaluation is presented in Chapter 7.

To increase the readability of this thesis we decided to put some topics in appendices. Especially for Dijkstra’s algorithm many data structures that implement a priority queue are developed over the years. In Appendix B we provide an overview of several of these methods. In the main part of this thesis the focus is on shortest path problems where only a single source is involved. The so-called all-pairs shortest path problem is the topic of Appendix C. In that appendix we also describe some theory about the shortest path problem where negative arc costs are involved. The similarities between some modern preprocessing techniques and earlier work in the field of Temporal Constraint Satisfaction Problem are discussed as well. Finally, in Appendix D we briefly describe the time-dependent shortest path problem, where the arc costs in the graph are considered to be time-dependent.

In this thesis, we focus on optimal shortest path and we therefore do not discuss approximate shortest path algorithms. A detailed study on approximate shortest paths is provided by Sommer [183]. Approximate shortest paths can be useful in traffic simulations where a large number of shortest paths has to be determined. Sommer observed that in the real world, drivers rarely use the exact shortest path. Hence, approximate shortest paths that can be determined fast can be quite useful for such simulations.
In this chapter, we give a formal description of algorithms and running times. Subsequently, we present some basic notation and definitions. Finally, we provide a mathematical model of the SPP.

2.1 Algorithms and running times

Knuth [123] starts his famous book series ‘The Art of Computer Programming’ with the historical background of the word *algorithm*. He then states that an algorithm is not only a finite set of rules that gives a sequence of operations for solving a specific type of problem, but that it has also five more characteristics:

- **Finiteness.** An algorithm must always terminate after a finite number of steps.
- **Definiteness.** Each step of an algorithm must be unambiguously specified for each case.
- **Input.** An algorithm has zero or more inputs (quantities that are entered, either initially or dynamically).
- **Output.** An algorithm has one or more outputs (quantities that have a specified relation to the input).
- **Effectiveness.** The operations of an algorithm must all be sufficiently basic to be executed in a finite amount of time.

In this section, we will mathematically describe an algorithm and its running time.

2.1.1 Alphabet

We follow [173] in providing a formal definition of an algorithm. First, we define the objects *symbols* and *strings*. 
**Definition 2.1** Let $\Sigma$ be a finite set (often $\Sigma = \{0,1\}$). $\Sigma$ is called the alphabet and its elements are called symbols or letters.

**Definition 2.2** An ordered, finite sequence of symbols from an alphabet $\Sigma$, is called a string (of symbols) or a word.

**Definition 2.3** Let $\Sigma$ be an alphabet. $\Sigma^*$ symbolizes the collection of all strings of symbols in $\Sigma$.

**Definition 2.4** Let $\Sigma$ be an alphabet. Let $z \in \Sigma^*$ be a string (word). The size of $z$ \((\text{size}(z))\) is the number of its symbols (letters).

Based on $\Sigma^*$, the notion of a problem can be defined:

**Definition 2.5** Let $\Pi \subseteq \Sigma^* \times \Sigma^*$, where $\Sigma^*$ is the collection of all strings of symbols of some alphabet $\Sigma$. A (search) problem is to find an output string $y$, given an input string $z \in \Sigma^*$, such that $(z,y) \in \Pi$, or to find that no such string $y$ exists.

### 2.1.2 Turing machine

Turing [192] introduced the concept of a computing machine. Nowadays, we call such a machine a Turing machine. The Turing machine mechanically operates on a tape. The tape contains symbols that the machine can read and write, one symbol at a time, using a tape head. The tape head can be moved either to the left or to the right. The operation of the machine is fully determined by a certain transition function $\delta$. More formally:

**Definition 2.6** A Turing machine is defined by the ‘7-tuple’ $(S, \Sigma, \Gamma, \delta, s_0, s_a, s_r)$. Here, $S$ is a set of states, $\Sigma$ is an alphabet and $\Gamma = \Sigma \cup \{\_\}$ is the alphabet together with a certain blank symbol. There are three special predefined states: the initial state $s_0 \in S$, the accepted state $s_a \in S$ and the rejected state $s_r \in S$. The transition function $\delta : S \times \Gamma \rightarrow S \times \Gamma \times \{\text{left, right}\}$ describes the actions that are executed when a certain symbol is read from the tape, while the machine is in a certain state. The actions are: write a predefined symbol to the tape, let the machine enter a predefined state and move the tape head either \text{left} or \text{right}. If either state $s_a$ or state $s_r$ is reached, the process terminates.

In Figure 2.1, a Turing machine is visualized. The transition function $\delta$ is not included in this figure.

**Definition 2.7** A universal Turing machine is a Turing machine that can simulate any regular type of Turing machine on any input. This is achieved by reading from its own tape both the description of the machine to be simulated, as well as the input of that machine.
2.1 Algorithms and running times

2.1.3 Algorithms

Now that we formally described a problem (Definition 2.5) and a universal Turing machine (Definition 2.7), we can formalize the notion of an algorithm.

**Definition 2.8** Let $\Sigma$ be an alphabet and let $\Pi$ be a problem. An algorithm to solve problem $\Pi$ can be defined as a finite string $A \in \Sigma^*$, which for any instance $z$ of $\Pi$ when giving the string $(A, z)$ to a universal Turing machine stops the machine after a finite number of steps, while either delivering a string $y$ with $(z, y) \in \Pi$, or no string at all should such a string $y$ not exist.

**Definition 2.9** Let $A$ be an algorithm for problem $\Pi$. The running time of algorithm $A$, for a certain problem instance $z$, is the number of moves the tape head of a universal Turing machine makes after having been given the input $(A, z)$.

**Definition 2.10** Let $A$ be an algorithm for problem $\Pi$. The running time function of algorithm $A$ is a function $f: \mathbb{Z}_+ \rightarrow \mathbb{Z}_+$:

$$f(n) := \max_{\{z \in \Sigma^*: \text{size}(z) \leq n\}} \{\text{running time of } A \text{ for input } z\}.$$

2.1.4 Random access machine

In Definition 2.9, the running time of an algorithm is the number of moves of the tape head of a universal Turing machine, running that algorithm. It is unclear how many moves of the tape head are needed, either to interpret the description and the input (See Definition 2.7) or to execute some basic operations like addition and multiplication.

To overcome this difficulty in computer science, the concept of a random access computer is widely used.

**Definition 2.11** A program is a numbered list of instructions.
Definition 2.12 [174] Let $\Sigma = \{0, 1\}$ be an alphabet. A random access machine (RAM) has a finite set of variables (strings in $\Sigma^*$ with a fixed size $n$) $r_0, r_1, \ldots, r_m$ and one array $z$ of length depending on the input. Each array entry is a string in $\Sigma^*$ with size $n$. Initially, the so-called program counter $r_0 = 0$, each so-called register $r_i = 0$ ($i = 1, 2, \ldots, m$) and $z$ contains the input. A program is executed on the RAM. The possible program statements are the elementary arithmetic operations, as shown in Table 2.1. The instructions of the program are numbered $0, 1, \ldots, t$. The program stops as soon as the program counter $r_0 > t$ and the contents of the array $z$ are returned as output.

Table 2.1: Instructions on a RAM ($i, j, k \in \{1, 2, \ldots, m\}$)

<table>
<thead>
<tr>
<th>Instruction</th>
</tr>
</thead>
<tbody>
<tr>
<td>$r_i \leftarrow r_j$</td>
</tr>
<tr>
<td>$z(r_j) \leftarrow r_i$</td>
</tr>
<tr>
<td>$r_i \leftarrow r_j + r_k$</td>
</tr>
<tr>
<td>$r_i \leftarrow r_j - r_k$</td>
</tr>
<tr>
<td>$r_i \leftarrow r_j r_k$</td>
</tr>
<tr>
<td>$r_i \leftarrow r_j/r_k$</td>
</tr>
<tr>
<td>$r_i \leftarrow r_i + 1$</td>
</tr>
<tr>
<td>$r_i \leftarrow 1$ if $r_j &gt; 0$ and $r_i \leftarrow 0$ otherwise</td>
</tr>
</tbody>
</table>

An instruction on a RAM can read several entries simultaneously, perform arithmetic operations and store their outcomes in array positions, prescribed by the instruction. This specific characteristic is indicated by the term ‘random access’ in ‘random access machine’: the machine has access, at any time, to the data in any position. This is clearly in contrast with the Turing machine, which can only move the tape head to adjacent positions.

In Figure 2.2 a RAM is visualized.

The RAM’s equivalent of the universal Turing machine is called a random access stored program machine.

Definition 2.13 A random access stored program (RASP) machine is a random access machine with its program and input data stored in its registers.
Definition 2.14 The running time of a program on a RASP is the total number of instructions executed.

Definition 2.15 The running time function of a program on a RASP is a function \( g : \mathbb{Z}_+ \rightarrow \mathbb{Z}_+ \):

\[
g(n) := \max_{z|z| \leq n} \{\text{running time of the program for input } z\}.
\]

From Definition 2.15, it is clear that the running time is based on a worst-case input. From here on, we use the term running time to indicate the ‘running time function of a program’ on a RASP. Furthermore, we use the term ‘algorithm’ to denote a (computer) program.

2.1.5 Asymptotic analysis

In 1894, Bachmann [6] introduced the \( O \)-notation. According to [98], this notation was made popular in subsequent years by Landau and others. Thus, the ‘big Oh’ symbol is also called the Landau symbol. We use the definition of [35]:

Definition 2.16 For a given function \( g(n) \), we denote by \( O(g(n)) \) the set of functions

\[
O(g(n)) = \{p(n) : \exists c > 0, n_0 > 0 \text{ such that } 0 \leq p(n) \leq cg(n) \forall n \geq n_0\}.
\]

To indicate that a function \( f(n) \) is a member of \( O(g(n)) \), we write \( f(n) = O(g(n)) \). If \( f(n) = O(g(n)) \), then \( g(n) \) is an asymptotic (for sufficiently large \( n, n \geq n_0 \)) upper bound of \( f(n) \). In Definition 2.16, we presented the \( O \)-notation for functions in one variable \( n \). We can easily adapt that notation to situations where more input variables are needed.

Sometimes we have a tight bound on \( f(n) \). We then use the \( \Theta \) notation:

Definition 2.17 For a given function \( g(n) \), we denote by \( \Theta(g(n)) \) the set of functions

\[
\Theta(g(n)) = \{p(n) : \exists c_1 > 0, c_2 > 0, n_0 > 0 \text{ such that } 0 \leq c_1 g(n) \leq p(n) \leq c_2 g(n) \forall n \geq n_0\}.
\]

To indicate that a function \( f(n) \) is a member of \( \Theta(g(n)) \), we write \( f(n) = \Theta(g(n)) \). If \( f(n) = \Theta(g(n)) \), then \( g(n) \) is an asymptotic tight bound of \( f(n) \).

From Definitions 2.16 and 2.17, it is obvious that \( \Theta(g(n)) \subseteq O(g(n)) \).
2.1.6 Parameter balancing

Sometimes the worst-case bound for an algorithm takes the form $O(f(n, m, k) + g(n, m, k))$, where $k$ is a tunable parameter. If both $f(n, m, k)$ and $g(n, m, k)$ are monotonically increasing in $k$, the optimal value for $k$ is clearly the minimum feasible value for $k$. On the other side, if both $f(n, m, k)$ and $g(n, m, k)$ are monotonically decreasing in $k$, the optimal value for $k$ is the maximum feasible value for $k$. In all other situations the optimal value for $k$ might be hard to determine. A technique called parameter balancing [3] can be used to determine a value for $k$ such that the resulting complexity is not larger than twice the minimum possible.

**Proposition 2.18** Let a worst-case bound $O(f(n, m, k) + g(n, m, k))$ be given where $n, m$ are problem characteristics and where $k$ is a tunable parameter. For any feasible value of $k$, we assume $f(n, m, k) \geq 0$ and $g(n, m, k) \geq 0$. Furthermore, $f(n, m, k)$ is monotonically increasing in $k$ and $g(n, m, k)$ is monotonically decreasing in $k$. Setting $k = k^*$ such that $f(n, m, k^*) = g(n, m, k^*)$, results in a complexity that is less than or equal to twice the minimum complexity possible.

**Proof:** Let $\bar{k}$ be the value that actually minimizes the complexity. If $k^* \geq \bar{k}$, then:

$$f(n, m, \bar{k}) \leq f(n, m, k^*) = g(n, m, k^*) \leq g(n, m, \bar{k}).$$

Thus,

$$f(n, m, \bar{k}) + g(n, m, \bar{k}) \geq g(n, m, \bar{k}) \geq g(n, m, k^*) = \frac{f(n, m, k^*) + g(n, m, k^*)}{2}.$$

Similarly, if $k^* < \bar{k}$ we have:

$$f(n, m, \bar{k}) \geq f(n, m, k^*) = g(n, m, k^*) \geq g(n, m, \bar{k}).$$

From which follows:

$$f(n, m, \bar{k}) + g(n, m, \bar{k}) \geq f(n, m, \bar{k}) \geq f(n, m, k^*) = \frac{f(n, m, k^*) + g(n, m, k^*)}{2}.$$

We conclude that choosing the value $k^*$ in such a way that $f(n, m, k^*) = g(n, m, k^*)$ results in a complexity less than or equal to twice the minimum complexity possible. \qed

2.1.7 Amortized analysis

Assume that a certain operation costs $O(g(n))$ time in the worst case. If that operation is executed $n$ times, iteratively, the worst-case bound is $O(n g(n))$. However, there are situations for which it is impossible that each operation in a particular sequence takes the worst-case time. Multiplying the worst-case time of the operation
by the number of operations in the sequence can easily result in a pessimistic bound. In such situations, the so-called amortized analysis can be used.

Instead of determining a worst-case bound for an individual operation, in an amortized analysis a worst-case bound for any sequence of operations of length $n$ is determined. Based on that bound, amortized costs are derived for the individual operations. Since amortized bounds are specified as an average bound for an individual operation, one should carefully distinguish amortized bounds from worst-case bounds. If an operation runs in $O(1)$ amortized time, it is meant that any sequence of $n$ operations runs in $O(n)$, although any particular operation may take $O(n)$ time itself.

There are three common ways in which an amortized analysis is performed [35]:

- **The aggregate method.** An upper bound $T(n)$ on the time needed to execute any sequence of $n$ operations, is determined. The amortized cost per operation is taken as $T(n)/n$. Note that from this analysis, one amortized cost is derived for possible different operations.

- **The accounting method (a banker’s view).** A cost (time) $\hat{c}_i$ is charged to operation $i$ as if the RAM (on which the operations are executed) is coin-operated. This pre-determined charge does not necessarily correspond to the actual time $c_i$ needed to execute operation $i$. It is possible that the operation will complete in less time than the time charged. In that case, some positive amount of credit is left ($\hat{c}_i - c_i$). It is also possible that the operation will need more time than the time charged, in which case the operation can be paid for by using previously accumulated credit. As long as the pre-determined charges (which in fact are the amortized times) are chosen to prevent going into debt, the amortized times will be an upper bound on the actual time for any sequence of operations:

  $$\sum_{i=1}^{n} \hat{c}_i \geq \sum_{i=1}^{n} c_i, \forall n \in \mathbb{N}, n \geq 1$$

Examples of this type of analysis can be found in [25, 109, 110].

- **The potential method (a physicist’s view).** In this method, the ‘prepaid’ work of the accounting method is represented as ‘potential energy’, or simply the ‘potential’. The potential is associated with the sequence as a whole rather than with specific operations. Consider a sequence in which $n$ operations are performed. We start with an initial state $D_0$. Let $c_i$ be the actual cost of the $i^{th}$ operation in the sequence and let $D_i$ represent the state after applying the $i^{th}$ operation on state $D_{i-1}$. A potential function $\Phi$ maps each state $D_i$ to a real number $\Phi(D_i)$. The amortized cost $\hat{c}_i$ of the $i^{th}$ operation is then defined by:

  $$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}).$$
For the whole sequence, we have:

\[ \sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) = \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0). \]

If a potential function \( \Phi \) can be found such that \( \Phi(D_n) \geq \Phi(D_0) \), then the sum of the amortized costs provides an upper bound to the actual cost for the whole sequence. The number \( n \) of operations in a sequence is not always known in advance, thus \( D_n \) might be unknown at the start. To ensure that we always ‘pay in advance’ (like the accounting method) a potential function \( \Phi \) should be used, for which \( \Phi(D_i) \geq \Phi(D_0) \) for all \( i (1 \leq i \leq n) \). Examples of this type of analysis can be found in [73, 181, 187].

Although the ideas behind amortized analysis were used earlier, they were first formalized by Tarjan [187].

### 2.2 Notation and definitions

Before we provide a mathematical model for the SPP, we present some basic notations and definitions, following [174] and [3].

#### 2.2.1 Graphs

First, we give a definition for a graph:

**Definition 2.19** A graph or undirected graph is a pair \( G = (V, E) \), where \( V \) is a finite set of nodes and \( E \) is a family of unordered pairs of nodes. The elements of \( E \) are called the edges of \( G \).

If \( \{v, w\} \in E \), nodes \( v \) and \( w \) are called adjacent or connected. Formally, \( E \) is a family rather than a set, because the same unordered pair of nodes may occur several times in \( E \). If a pair \( \{v, w\} \) occurs more than once in \( E \), \( \{v, w\} \) is called a multiple edge. Edges of the form \( \{v, v\} \) are called loops.

**Definition 2.20** An undirected graph \( G = (V, E) \) without loops and multiple edges, is called simple.

In a simple undirected graph \( G = (V, E) \), we have \( E \subseteq \{\{v, w\} \mid v, w \in V, v \neq w\} \). We can refer to edge \( \{v, w\} \) without ambiguity, since there are no multiple edges. In subscripts, we will shortly write \( vw \) for edge \( \{v, w\} \).

**Definition 2.21** A directed graph, or digraph, is a pair \( G = (V, A) \), where \( V \) is a finite set of nodes and \( A \) is a family of ordered pairs of nodes. The elements of \( A \) are called the arcs of \( G \).
2.2 Notation and definitions

If \((v, w) \in \mathcal{A}\), nodes \(v\) and \(w\) are called \emph{adjacent} or \emph{connected}. Similar to the undirected version, \(\mathcal{A}\) is a family rather than a set, because the same ordered pair of nodes may occur several times in \(\mathcal{A}\). If a pair \((v, w)\) occurs more than once in \(\mathcal{A}\), \((v, w)\) is called a \emph{multiple} arc. Arcs of the form \((v, v)\) are called \emph{loops}.

Definition 2.22 A digraph \(\mathcal{G} = (\mathcal{V}, \mathcal{A})\) without loops and multiple arcs, is called simple.

In a simple digraph \(\mathcal{G} = (\mathcal{V}, \mathcal{A})\), we have \(\mathcal{A} \subseteq \{(v, w) \mid v, w \in \mathcal{V}, v \neq w\}\). We can refer to arc \((v, w)\) without ambiguity since there are no multiple arcs. In subscripts, we will shortly write \(vw\) for \((v, w)\).

Each directed graph \(\mathcal{G} = (\mathcal{V}, \mathcal{A})\) has an \emph{underlying undirected simple graph}, which is the graph \(\mathcal{G} = (\mathcal{V}, \mathcal{E})\), where \(\{v, w\} \in \mathcal{E}\) if \((v, w) \in \mathcal{A}\) or \((w, v) \in \mathcal{A}\).

2.2.2 Paths and cycles

We define a \emph{path} as follows:

Definition 2.23 Let \(\mathcal{G} = (\mathcal{V}, \mathcal{E})\) be an undirected graph. A \emph{path} or undirected path \(P\) is a sequence of nodes and edges \((v_0, e_1, v_1, \ldots, e_k, v_k)\), where \(k \geq 0\), \(v_i \in \mathcal{V}\) (for \(i = 0, 1, \ldots, k\)) and \(e_i \in \mathcal{E}\) connects \(v_{i-1}\) and \(v_i\) (for \(i = 1, 2, \ldots, k\)). If \(v_i \neq v_j\) for all \(0 \leq i < j \leq k\), we say that the path is simple.

Definition 2.24 Let \(P = (e_0, v_1, v_2, \ldots, e_k, v_k)\) be a path on the graph \(\mathcal{G} = (\mathcal{V}, \mathcal{E})\). If \(v_0 = v_k\) the path is called a \emph{closed path} or a \emph{cycle}. \(P\) is called a simple cycle if \(v_0 = v_k\), \(k \geq 1\) and \(v_i \neq v_j\) for all \(1 \leq i < j \leq k\).

For a digraph \(\mathcal{G} = (\mathcal{V}, \mathcal{A})\) we use similar definitions:

Definition 2.25 Let \(\mathcal{G} = (\mathcal{V}, \mathcal{A})\) be a digraph. A \emph{directed path} \(P\) is a sequence of nodes and arcs \((v_0, a_1, v_1, \ldots, a_k, v_k)\), where \(k \geq 0\), \(v_i \in \mathcal{V}\) (for \(i = 0, 1, \ldots, k\)) and \(a_i \in \mathcal{A}\) connects \(v_{i-1}\) and \(v_i\) (for \(i = 1, 2, \ldots, k\)). If \(v_i \neq v_j\) for all \(0 \leq i < j \leq k\), we say that \(P\) is a simple directed path.

Definition 2.26 Let \(P = (a_0, v_1, v_2, \ldots, a_k, v_k)\) be a directed path on the digraph \(\mathcal{G} = (\mathcal{V}, \mathcal{A})\). If \(v_0 = v_k\) the path is called a \emph{closed directed path} or a \emph{directed cycle}. \(P\) is called a \emph{simple directed cycle} if \(v_0 = v_k\), \(k \geq 1\) and \(v_i \neq v_j\) for all \(1 \leq i < j \leq k\).

From this point onward, where it is clear from the context that the directed path is defined on a digraph, we will simply write \emph{path} instead of \emph{directed path}. Node \(v_0\) is called the \emph{origin} and node \(v_k\) is called the \emph{destination} of the path \(P\). A path from origin \(s\) to destination \(t\) is called an \emph{s-t path}, denoted by \(P_{st}\).

Sometimes, we need to know whether or not there exists a \(v\)-\(w\) path in a graph. We use the following terminology:
Definition 2.27 Let $G = (V, E)$ be a graph. Node $w$ is reachable from node $v$ if there exists a $v$-$w$ path in $G$.

For a directed graph, a similar terminology is used:

Definition 2.28 Let $G = (V, A)$ be a digraph. Node $w$ is reachable from node $v$ if there exists a $v$-$w$ path in $G$.

2.2.3 Connectivity and components

Next, we define connectivity:

Definition 2.29 A graph $G = (V, E)$ is connected if, for any two nodes $v$ and $w$, a $v$-$w$ path exists.

Definition 2.30 Let $G = (V, E)$ be a graph. The graph $G' = (V', E')$ is called a subgraph of $G$, if $V' \subseteq V$ and $E' \subseteq E$.

Definition 2.31 Let $G' = (V', E')$ be a subgraph of the graph $G = (V, E)$. $G'$ is called induced (or full) if $(v, w) \in E'$ for any $(v, w) \in E$ with $v, w \in V'$.

Definition 2.32 A maximal (with respect to the number of nodes) connected non-empty induced subgraph of $G = (V, E)$ is called a connected component, or simply component.

We use similar definitions for a directed graph:

Definition 2.33 A digraph $G = (V, A)$ is called strongly connected if, for any two nodes $v$ and $w$, a directed $v$-$w$ path exists. A digraph $G = (V, A)$ is called weakly connected when the underlying undirected graph of $G$ is connected.

Definition 2.34 Let $G = (V, A)$ be a digraph. The digraph $G' = (V', A')$ is called a subgraph of $G$, if $V' \subseteq V$ and $A' \subseteq A$.

Definition 2.35 Let $G' = (V', A')$ be a subgraph of the digraph $G = (V, A)$. $G'$ is called induced (or full) if $(v, w) \in A'$ for any $(v, w) \in A$ with $v, w \in V'$.

Definition 2.36 A maximal (with respect to the number of nodes) strongly connected non-empty induced subgraph of $G = (V, A)$ is called a strongly connected component, or strong component.
2.2 Notation and definitions

2.2.4 Forward and backward star

To simplify notation, we introduce a notation for the incoming and outgoing arcs of a set of nodes, also called backward star and forward star, respectively:

Definition 2.37 The forward star of a subset of nodes $B \subseteq V$, denoted by $\delta^+(B)$, consists of the outgoing arcs of $B$:

$$\delta^+(B) = \{ (v, w) \in A \mid v \in B, w \notin B \}.$$  

N.B. If $B$ consists of only one node $v$, we write $\delta^+(v)$.

Definition 2.38 The backward star of a subset of nodes $B \subseteq V$, denoted by $\delta^-(B)$, consists of the incoming arcs of $B$:

$$\delta^-(B) = \{ (v, w) \in A \mid w \in B, v \notin B \}.$$  

N.B. If $B$ consists of only one node $v$, we write $\delta^-(v)$.

2.2.5 Arc cost and path length

Let $G = (V, A)$ be a digraph. We assume that a function $C : A \to \mathbb{R}$ exists, which assigns a cost $C((v, w))$ to any arc $(v, w) \in A$. We write the cost $C((v, w))$ of arc $(v, w)$ also as $C(v, w)$ or $c_{vw}$.

Let $s, t \in V$ and $P$ be an $s$-$t$ path. The length $\ell(P)$ of $P$ is defined by

$$\ell(P) := \sum_{(v, w) \in P} c_{vw}. \quad (2.1)$$

The distance between $s$ and $t$, denoted by $d(s, t)$, is defined as the minimum length of all $s$-$t$ paths. Let $P_{st}$ be the set of all $s$-$t$ paths:

$$d(s, t) = \min \{ \ell(P) \mid P \in P_{st} \}. \quad (2.2)$$

Definition 2.39 Let $G = (V, A)$ be a digraph, $s, t \in V$ and $P$ an $s$-$t$ path. $P$ is a shortest $s$-$t$ path if and only if $\ell(P) = d(s, t)$.

Definition 2.40 Let $P = (v_0, a_1, v_1, \ldots, a_k, v_k)$ be a directed path on the digraph $G = (V, A)$. For any $i, j$ such that $0 \leq i \leq j \leq k$, the sequence $(v_i, a_{i+1}, v_{i+1}, \ldots, v_j)$ is called a subpath of $P$.

Theorem 2.41 Let $G = (V, A)$ be a digraph, $s, t \in V$ and $P_{st}$ be a shortest $s$-$t$ path. Any subpath of $P$ is a shortest path itself.
Proof: We prove this by contradiction. Let $P_{st} = (v_0 = s, a_1, v_1, \ldots, a_k, v_k = t)$ be a shortest $s$-$t$ path and $P_{v_i,v_j}$ be a subpath of $P_{st}$. We decompose $P_{st}$ as follows: $P_{st} = P_{s,v_i} + P_{v_i,v_j} + P_{v_j,t}$ with length $\ell(P_{s,v_i}) + \ell(P_{v_i,v_j}) + \ell(P_{v_j,t}) = \ell(P_{st}) = d(s,t)$. Assume there exists a path $P'_{v_i,v_j}$ with $\ell(P'_{v_i,v_j}) < \ell(P_{v_i,v_j})$. The $s$-$t$ path that is composed by concatenating $P_{s,v_i} + P'_{v_i,v_j} + P_{v_j,t}$ has length $\ell(P_{s,v_i}) + \ell(P'_{v_i,v_j}) + \ell(P_{v_j,t}) < \ell(P_{st})$, which contradicts that $P_{st}$ is a shortest $s$-$t$ path. Hence, the subpath $P_{v_i,v_j}$ of $P_{st}$ is a shortest path. 

The arc costs may have different interpretations. In a real road network for example, the arc lengths usually represent the length of the corresponding road. The solution of SPP then literally gives the length of the shortest path. Another possibility is to set the arc lengths equal to the expected travel time of the corresponding road. The solution of SPP then gives the shortest expected travel time and a corresponding path from origin to destination.

Sometimes we use the diameter of a graph:

Definition 2.42 Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a connected graph. The diameter of $\mathcal{G}$ is the maximum shortest path distance between any pair of nodes.

2.2.6 Shortest Path Tree

Definition 2.43 Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph. If $\mathcal{G}$ contains no cycles, $\mathcal{G}$ is called a forest. If moreover $\mathcal{G}$ is connected, then $\mathcal{G}$ is called a tree. A connected subgraph of a tree $\mathcal{G}$ is called a subtree of $\mathcal{G}$.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a graph. A subset $\mathcal{F}$ of $\mathcal{E}$ is called a forest if $(\mathcal{V}, \mathcal{F})$ is a forest and a spanning tree if $(\mathcal{V}, \mathcal{F})$ is a tree.

Definition 2.44 Let $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ be a digraph. $\mathcal{G}$ is a directed tree if $\mathcal{G}$ is weakly connected and $\mathcal{G}$ has no undirected cycles. In other words; $\mathcal{G}$ is a directed tree if its underlying undirected graph is a tree.

Definition 2.45 Let $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ be a digraph. A node $v \in \mathcal{V}$ with $\delta^-(v) = \emptyset$ is called a source. A node $v \in \mathcal{V}$ with $\delta^+(v) = \emptyset$ is called a sink.

Proposition 2.46 Let $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ be a digraph, with $|\mathcal{V}| \geq 1$. If $\mathcal{G}$ contains no directed cycles, then $\mathcal{G}$ has at least one sink and at least one source.

Proof: Let $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ be a digraph without a source. So every node $v \in \mathcal{V}$ has at least one incoming arc. Start from any node $w \in \mathcal{V}$ and follow one arc $(v,w) \in \delta^-(w)$. Again follow some arc $(u,v) \in \delta^-(v)$ and repeat this $|\mathcal{V}| + 1$ times. Since $\mathcal{G}$ has $|\mathcal{V}|$ nodes, we must have visited at least one node more than once. Thus, we find a directed cycle. We conclude that a digraph $\mathcal{G}$ without any directed cycles has at least one source.
In a similar way, we can prove that a digraph $\mathcal{G}$ without any directed cycle has at least one sink.

**Definition 2.47** Let $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ be a directed tree. If $\mathcal{G}$ has exactly one source, then $\mathcal{G}$ is called a rooted tree or an outtree rooted at node $s$.

**Definition 2.48** A shortest path tree rooted at $s$ is a rooted tree such that the directed path from $s$ to $v$ is a shortest path from $s$ to $v$ for all $v \in \mathcal{V}$.

The one-to-all shortest path problem consists of finding a shortest path tree rooted at any given node $s$.

### 2.2.7 Assumptions

The following assumptions are imposed on the SPP:

- The graph is strongly connected.
- We assume all arc costs to be nonnegative: $c_{vw} \geq 0$ for all $(v, w) \in \mathcal{A}$.

In related literature, another assumption is commonly stated: the graph does not contain a directed cycle with negative length. This assumption is naturally fulfilled due to assumption of nonnegative arc lengths.

For some algorithms, as we will see later, we need the additional assumption that all arc lengths are integer. This is not a restrictive assumption in cases where the graph is a road map and the arc lengths denote distances or travel times, since we can redefine the arc’s characteristics by multiplying the rational arc lengths by a suitably large enough number. In other graphs it might happen that the scaling factor needed, to get suitable integer arc lengths, leads to extremely large arc costs. In such cases it is possible that some algorithms, based on the assumption that all arc lengths are integer, cannot be used.

### 2.3 Formulation as a Minimum Cost Flow Problem

The Minimum Cost Flow Problem (MCFP) is the core linear network flow problem of which many other linear network flow problems are special cases. Therefore, we first define the MCFP together with some basic definitions for network flow problems.

Let $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ be a digraph as defined in Definition 2.21. Each arc $(v, w) \in \mathcal{A}$ has a capacity $u_{vw}$, denoting the maximum possible flow on the arc, and a lower bound $\ell_{vw}$, which denotes the minimum amount that must flow on the arc. Furthermore, we associate with each node $v \in \mathcal{V}$ an integer $b_v$; if $b_v > 0$, then node $v$ is a supply node; if $b_v < 0$, then node $v$ is a demand node; if $b_v = 0$, then node $v$ is a transshipment node. The MCFP now deals with finding through the graph a least-cost flow that
satisfies both the flow bounds and the flow demands from available supplies at other nodes. This optimization problem can now be formulated as follows:

\begin{align*}
\text{minimize} & \quad \sum_{(v,w) \in A} c_{vw} x_{vw} \\
\text{subject to} & \quad \sum_{\{w|(v,w) \in \delta^+(v)\}} x_{vw} - \sum_{\{w|(w,v) \in \delta^-(v)\}} x_{wv} = b_v, \quad \forall v \in V, \quad (2.4) \\
& \quad \ell_{vw} \leq x_{vw} \leq u_{vw}, \quad \forall (v,w) \in A, \quad (2.5)
\end{align*}

where \( x_{vw}, (v,w) \in A \) is called the \textit{flow vector} \( x \). We refer to \( x_{vw} \) as the flow on the arc \((v,w)\). The equations (2.4) are called the \textit{balance equations}. The constraints (2.5) are called the \textit{capacity constraints}.

### 2.3.1 Formulation of the SSSD-SPP

The single-source single-destination shortest path problem (SSSD-SPP) can be formulated as a MCFP, where the supply vector \( b \) is given by \( b_s = 1, b_t = -1, b_v = 0, \forall v \neq s,t \), except that we want the flow vector to only take binary values: \( x_{vw} \in \{0,1\} \). This way, we can represent a path \( P \) from \( s \) to \( t \) as a path flow \( x \), defined by

\[
x_{vw} = \begin{cases} 
1 & \text{if } (v,w) \text{ belongs to } P, \\
0 & \text{otherwise.}
\end{cases} \quad (2.6)
\]

We can see that a path \( P \) from \( s \) to \( t \) is shortest, if and only if this path flow \( x \) is an optimal solution to the MCFP for SSSD-SPP.

Since we have to minimize a linear function over vectors that may only take integer values (even only the binary values 0 and 1), this problem is in fact an \textit{integer linear optimization problem}. We can write the above mentioned formulation for SSSD-SPP, as follows:

\[
\min \{ c^T x \mid Bx = b; \ 0 \leq x \leq 1 \}.
\]

The matrix \( B \) then is a \(|V| \times |A|\) matrix, which is called the \textit{incidence matrix} of \( G \). The columns and the rows represent the arcs and nodes of \( G \) respectively. Each column has exactly one \(-1\) and one \(1\); for arc \((v,w)\), the \(1\) corresponds to node \( v \) and the \(-1\) corresponds to node \( w \). Also, each square submatrix has a determinant equal to 0, +1, or \(-1\). Such a matrix is called \textit{totally unimodular}. It is known that for a totally unimodular matrix \( B \), each vertex of the polytope \( \{x \mid Bx = b, x \geq 0\} \) is integer when \( b \) is integer (see for example [193]). This results in a minimum cost flow problem formulation that, without the binary constraint \( x_{vw} \in \{0,1\} \), still has
optimum solutions with integer values for the flow vector $x$. Because of the balance equations of SSSD-SPP, an integer flow vector can only be valued binary. Hence, we may relax the constraint $0 \leq x \leq 1$ to the nonnegativity constraint $x \geq 0$.

The minimum cost flow formulation for SSSD-SPP is thus given by (2.3), (2.4) and (2.5) with

$$
\begin{align*}
    b_s &= 1, & b_t &= -1, & b_v &= 0 \forall v \neq s, t, \\
    \ell_{vw} &= 0, & u_{vw} &= \infty.
\end{align*}
$$

### 2.3.2 Formulation of the MSMD-SPP

Suppose we are interested in the shortest path from a set $S \subset V$ of origin nodes to a set $T \subset V$ of several destinations. Such a multiple-source multiple-destination shortest path problem (MSMD-SPP) can be seen as a collection of $|S|$ single-source multiple-destination shortest path problems (SSMD-SPP).

For each $s \in S$, the minimum cost flow formulation for the corresponding SSMD-SPP is given by (2.3), (2.4) and (2.5) with (2.8) and

$$
    b_v = \begin{cases} 
        |T| & \text{if } v = s \text{ and } s \not\in T, \\
        |T| - 1 & \text{if } v = s \text{ and } s \in T, \\
        -1 & \text{if } v \in T\backslash\{s\}, \\
        0 & \text{otherwise.} 
    \end{cases}
$$

This way, we can represent the paths from $s$ to any destination $t \in T$, with a flow $x$ defined by

$$
    x_{vw} = \begin{cases} 
        > 0 & \text{if } (v, w) \text{ belongs to a path starting from } s \text{ to a node } t \in T, \\
        0 & \text{otherwise.}
    \end{cases}
$$

The SSMD-SPP formulation can be rewritten as a linear optimization problem:

$$
\min \{ c^T x \mid Bx = b; \ x \geq 0 \}.
$$

As stated in Section 2.3.1, it is not necessary to explicitly restrict $x$ to integer values, due to the fact that the matrix $B$ is totally unimodular and $b$ is integer.

### 2.4 The dual problem

Since the minimum cost flow formulation for SPP is a linear optimization problem, it has a related dual problem [74]. If the primal problem has an optimal solution, so does its dual, and at optimality their costs are equal.
The standard form of a linear program and its dual is given by:

**primal:** \( \min \{ c^T x \mid Bx = b, x \geq 0 \} \)

**dual:** \( \max \{ b^T p \mid B^T p \leq c \} \) \hfill (2.11)

Hence, the dual of the SPP defined by (2.3), (2.4) and (2.5), is equal to:

\[
\begin{align*}
\text{maximize} & \quad b^T p \\
\text{subject to} & \quad p_v - p_w \leq c_{vw} \quad \forall (v, w) \in A.
\end{align*}
\] \hfill (2.12)

The dual variables \( p_v \) introduced here are called **prices**.

**Definition 2.49** Given a price vector \( p \), the **reduced cost** \( c_{vw} \) of an arc \((v, w) \in A\) is given by

\[
c_{vw} := c_{vw} + p_w - p_v.
\]

With Definition 2.49, the dual feasibility (2.13) can be equivalently stated as \( c_{vw} \geq 0 \) for all \((v, w) \in A\).

### 2.4.1 Optimality conditions for the SPP

Most shortest path algorithms maintain and adjust a vector \((d_1, d_2, \ldots, d_{|V|})\), where each \( d_v \) is either a scalar or \( \infty \), which is called the **label** of node \( v \). The labels are related to the dual variables, the prices, by the following equality: \( p_v = -d_v \).

Shortest path algorithms are based on the simple optimality condition, given in the following theorem:

**Theorem 2.50** [17] Let \( d = (d_1, d_2, \ldots, d_{|V|}) \) be a vector satisfying

\[
d_w \leq d_v + c_{vw}, \quad \forall (v, w) \in A
\] \hfill (2.14)

and let \( P \) be a path that starts at node \( s \) and ends at node \( t \). If

\[
d_w = d_v + c_{vw}, \quad \forall (v, w) \in P,
\] \hfill (2.15)

then \( P \) is a shortest path from \( s \) to \( t \).

**Proof**: Let the path \( P = (v_1 = s, (v_1, v_2), v_2, \ldots, (v_{k-1}, v_k), v_k = t) \) satisfy equation (2.15). By adding this equation over all arcs of \( P \), the label of the last node of the path becomes

\[
d_t = d_{v_{k-1}} + c_{k-1,k} = d_{v_{k-2}} + c_{k-2,k-1} + c_{k-1,k} = \ldots = d_s + \sum_{(v,w) \in P} c_{vw}.
\]
Hence, the length of $P$ is equal to the difference of the labels of the nodes $d_t - d_s$. Now, if we do the same for any other path $P'$ starting at $s$ and ending at $t$, this path satisfies (2.14) and we have

$$d_t \leq d_s + \sum_{(v,w) \in P'} c_{vw}.$$ 

This shows that the length of $P'$ must be no less than $d_t - d_s$. Therefore, $P$ is a shortest path.

### 2.4.2 Complementary slackness conditions for the SPP

The conditions (2.14) and (2.15) of Theorem 2.50 are called the complementary slackness (CS) conditions for the SPP. In fact, these conditions are special cases of the general optimality condition (also called CS condition) to the equivalent MCFP, which in turn forms a special case of a corresponding CS condition for general linear programs.

The complementary slackness property can be derived from the duality theory. For any feasible vector $x$ and $y$ for the standard formulation of a linear program and its dual (2.11), we know that the relation $c^T x \geq y^T(Bx) = b^T y$, $x \geq 0$ and $Bx = b$ imply that $x^T c \geq x^T(B^Ty) = y^T Bx = y^T b$.

If $c^T x = y^T(Bx)$, then $c^T x = b^T y$ and $x$ and $y$ are optimal for their respective problems. Rewriting this equation yields the following complementary slackness condition:

$$x^T (c - B^Ty) = 0. \tag{2.16}$$

Thus, if we have a feasible flow vector $x$ and feasible price vector $p$ that satisfy the complementary slackness conditions, that is,

$$x_{vw}(c_{vw} - p_v + p_w) = 0, \forall (v, w) \in A, \tag{2.17}$$

then the pair $(x, p)$ is an optimal solution to both the primal as well as to the dual problem.

This gives a second proof for Theorem 2.50: let $d$ be a vector satisfying (2.14) and (2.15). With $p_v = -d_v$, (2.14) implies that $p$ is dual feasible. Furthermore, remember that if $x$ is a primal feasible flow vector, the flow vector satisfies (2.10). Since (2.15) holds, we see that $-p_w = -p_v + c_{vw}$ for all $(v, w) \in P$. Hence, $c_{vw} - p_v + p_w = 0$ for all $(v, w) \in A$ for which $x_{vw} \neq 0$, and so the complementary slackness condition (2.17) is satisfied. Thus, it follows that the pair $(x, d)$ gives an optimal solution. Therefore, the path corresponding to the flow vector $x$ is a shortest path.

N.B. If we identify $p_v$ with $-d_v$, the CS conditions for SPP (Section 2.3) take the form:
\[ p_v \leq c_{vw} + p_w, \quad \forall (v, w) \in A, \quad (2.18) \]
\[ p_v = c_{vw} + p_w, \quad \forall (v, w) \in P. \quad (2.19) \]

### 2.4.3 Bellman’s Equations

The optimality conditions of (2.14) are also called *Bellman’s conditions*.

When the Bellman’s conditions hold, the solution is dual feasible, as the conditions are equivalent to the restrictions (2.13) of the dual problem.

The *non-linear Bellman’s Equation system* is defined as (see [15]):

\[
\begin{align*}
    d_s &= 0, \\
    d_w &= \min_{(v, w) \in A} \{d_v + c_{vw}\}, \quad \forall w \neq s.
\end{align*}
\quad (2.20)
\]

Many shortest path algorithms can be interpreted as procedures for solving this equation system. It expresses that the shortest distance between \(s\) and \(t\) is the sum of the shortest distance between \(s\) and the node preceding \(t\) on the shortest path, plus the length of the arc connecting that particular node to \(t\).
Different methods for solving the SPP

In this chapter, we discuss several algorithms to calculate the shortest path from an origin $s$ to a set of destination nodes $T \subseteq V$.

### 3.1 Generic Shortest Path algorithm

The Generic Shortest Path algorithm is a prototype shortest path algorithm, which contains several interesting algorithms as special cases. The algorithm is based on the optimality conditions for the shortest path problem (2.14). Arcs that violate Bellman’s conditions (i.e. $d_w > d_v + c_{vw}$) are successively selected and the labels are set to such a value that the complementary slackness (CS) condition for that particular arc is satisfied. This continues until the CS condition is satisfied for all arcs and the shortest path from $s$ to all other nodes of $V$ is found.

Before we describe the Generic Shortest Path algorithm in detail, we first introduce a way to store a shortest path tree, as well as some subroutines.

#### 3.1.1 Keeping track of the shortest path tree

In many situations we not only calculate the length of the shortest path, but we need the nodes on the shortest path as well. For each node $v \in V$, we maintain a predecessor (or parent) $\pi_v$ that either points to another node or to $\text{NIL}$. At the end of a shortest path algorithm, the shortest path from origin $s$ to a destination node $t \in T$ can be visualized by Algorithm 3.1 (see [35]). At the end of a one-to-all shortest path algorithm that maintains a predecessor $\pi_v$ for all nodes $v \in V$, the graph $G_{\pi} = (V, A_{\pi})$ with $A_{\pi} = \{(\pi_v, v) \mid v \in V, \pi_v \neq \text{NIL}\}$ is a shortest path tree (see Definition 2.48).
Algorithm 3.1: \textsc{Print-Path}(s,t)

\begin{algorithm}
  \textbf{Require:} $\pi_t \neq \text{NIL}$ or $s = t$
  1: if $s \neq t$ then
  2: \textbf{Print-Path}(s,$\pi_t$)
  3: print $t$
\end{algorithm}

3.1.2 Two common subroutines

The algorithms in this chapter use two common subroutines, stated in Algorithms 3.2 and 3.3. Both are based on similar routines as described in [35].

Algorithm 3.2: \textsc{Initialize-Single-Source}($G$, $s$, $Q$)

\begin{algorithm}
  1: foreach $v \in V$ do
  2: \hspace{1em} $d_v \leftarrow \infty$
  3: \hspace{1em} $\pi_v \leftarrow \text{NIL}$
  4: \hspace{1em} $d_s \leftarrow 0$
  5: \hspace{1em} $Q \leftarrow \{s\}$
\end{algorithm}

The first subroutine (Algorithm 3.2) initializes the algorithm. After initialization, we have $\pi_v = \text{NIL}$ for all $v \in V$, $d_s = 0$ and $d_w = \infty$ for $v \in V \setminus \{s\}$. Most algorithms in this chapter maintain a candidate set $Q$ of nodes. After initialization, this candidate set contains the origin $s$ only.

Algorithm 3.3: \textsc{Label-Update}(v, w, Q)

\begin{algorithm}
  1: if $d_v + c_{vw} < d_w$ then
  2: \hspace{1em} $d_w \leftarrow d_v + c_{vw}$
  3: \hspace{1em} $\pi_w \leftarrow v$
  4: \hspace{1em} if $w \notin Q$ then
  5: \hspace{2em} $Q \leftarrow Q \cup \{w\}$
\end{algorithm}

The second subroutine (Algorithm 3.3) performs a test: if the shortest path to node $w$ found so far can be improved by the shortest path to node $v$ and the edge $(v, w)$, then $d_w$ and $\pi_w$ are updated. Furthermore, if $d_w$ is updated and $w$ is not yet in the candidate set $Q$, $w$ is added to $Q$. We sometimes refer to a \textsc{Label-Update}(v, w, Q) as scanning arc $(v, w)$.

3.1.3 Generic Shortest Path algorithm

Now that we have introduced the \textsc{Initialize-Single-Source} and \textsc{Label-Update} subroutines, we present the Generic Shortest Path algorithm [75]:

\begin{algorithm}
  \hspace{1em} if $d_v + c_{vw} < d_w$ then
  \hspace{2em} $d_w \leftarrow d_v + c_{vw}$
  \hspace{2em} $\pi_w \leftarrow v$
  \hspace{2em} if $w \notin Q$ then
  \hspace{3em} $Q \leftarrow Q \cup \{w\}$
\end{algorithm}
3.1 Generic Shortest Path algorithm

Algorithm 3.4: Generic Shortest Path \((G, s)\)

1: \textsc{Initialize-Single-Source}(\(G, s, Q\))
2: \textbf{repeat}
3: \hspace{1em} Select a pivot node \(v\) from \(Q\)
4: \hspace{1em} \(Q \leftarrow Q \setminus \{v\}\)
5: \hspace{1em} \textbf{foreach} \((v, w) \in \delta^+(v)\) \textbf{do}
6: \hspace{2em} \textsc{Label-Update}(v, w, Q)
7: \hspace{1em} \textbf{until} \(Q = \emptyset\)

Theorem 3.1 [19] At the end of each iteration of the Generic Shortest Path algorithm, the following conditions hold:

(i) if \(d_v < \infty\), then \(d_v\) is the length of some path that starts at \(s\) and ends at \(v\).
(ii) if \(v \notin Q\), then either \(d_v = \infty\) or \(d_w \leq d_v + c_{vw} \forall (v, w) \in \delta^+(v)\).

Proof: We prove part (i) by induction. At the end of the first iteration, the nodes \(v \in V\) with \(d_v < \infty\) are node \(s\) (\(d_s = 0\) is the length of the path containing only node \(s\)) and those for which \((s, v) \in A\) (\(d_v = c_{sv}\)). So (i) holds at the end of the first iteration. Now suppose that (i) holds at the end of an iteration in which node \(v\) was removed from \(Q\). Then \(d_v < \infty\), and (by the induction hypothesis) \(d_v\) is the length of some \(s-v\) path \(P\). When a label \(d_w\) is changed during the iteration, \(d_w\) is set to \(d_v + c_{vw}\), which is the length of the path consisting of \(P\) followed by arc \((v, w)\) and node \(w\). Thus, property (i) holds at the end of the iteration, completing the induction proof.

To prove (ii), note that for any \(v\), each time \(v\) is removed from \(Q\), the condition \(d_w \leq d_v + c_{vw}\) is satisfied for all \((v, w) \in \delta^+(v)\) due to the \textsc{Label-Update}(\(v, w, Q\))-subroutine. Up to the next entrance of \(v\) in \(Q\), \(d_v\) stays constant while the labels \(d_w\) with \((v, w) \in \delta^+(v)\) cannot increase, therefore preserving the condition \(d_w \leq d_v + c_{vw}\).

Theorem 3.2 [19] Let \(G\) be a strongly connected graph with nonnegative arc lengths. Then, the Generic Shortest Path algorithm terminates and upon termination for all \(v \in V\), \(d_v\) is the length of a shortest \(s-t\) path.

Proof: Firstly, we prove that the algorithm will terminate. In accordance with Theorem 3.1 part (i), each time a label \(d_v\) is decreased, \(d_v\) equals the length of an \(s-v\) path. The graph \(G\) does not contain cycles with negative length (because all arc lengths are nonnegative). Therefore, each time a label \(d_v\) is decreased the \(s-v\) path is a simple path. The number of different simple \(s-v\) paths is finite. Thus, each label \(d_v\) can only be decreased a finite number of times.

Secondly, we show that for all \(v \in V\), upon termination, \(d_v\) is the length of a shortest \(s-v\) path. Theorem 3.1 part (ii) implies that upon termination, since \(Q = \emptyset\), the inequalities (2.14) hold. Part (i) implies that upon termination, \(d_v\) is the length of
some path $P$ from $s$ to node $v$. The equalities (2.15) hold for $P$. By Theorem 2.50, we know $P$ is a shortest $s$-$v$ path.

**Theorem 3.3** [3] Consider the Generic Shortest Path algorithm 3.4 on a graph with arcs of integral (nonnegative) length. Let $C = \max\{c_{uw} | (v, w) \in A\}$. The number of iterations is $O(\delta^+ V |A|)$.

**Proof:** Under the assumption that all arcs have integral (nonnegative) length, the labels $d_v$, $v \in V$ are finite and satisfy the bounds $0 \leq d_v \leq C|V|$. Therefore, the algorithm updates any label $d_v$ at most $C|V|$ times because each update of $d_v$ decreases it by at least 1 unit. The number of executions of the Label-Update-subroutine is bounded by:

$$C|V| \sum_{v \in V} |\delta^+(v)| = C|V||A|. \quad (3.1)$$

Therefore the algorithm runs in $O(C|V||A|)$ time.

If $C = O(2^{|V|})$, the Generic Shortest Path algorithm on a graph with arcs of integral (nonnegative) lengths has running time $O(2^{|V|}|V||A|)$. On the other hand, if $C$ is not exponentially large in terms of $|V|$ or $|A|$, the running time of the Generic Shortest Path algorithm on such a graph is polynomially bounded.

Let us now consider the general situation where the arc lengths are not necessarily integral.

**Theorem 3.4** [3] Consider the Generic Shortest Path algorithm 3.4 on a graph with arcs having nonnegative length. The number of iterations is $O(2^{|V|})$.

**Proof:** Each time the label $d_v$ of node $v$ is updated, it represents the length of a $P_{sv}$ path (see Theorem 3.1). Let $P_{sv}^1$ and $P_{sv}^2$ be two paths to node $v$, based on two different updates of $d_v$. Since $d_v$ is updated twice, we have $P_{sv}^1 \neq P_{sv}^2$. We define the sets $V^1 = \{u | u \in P_{sv}^1\}$ and the set $V^2 = \{u | u \in P_{sv}^2\}$ and we prove by contradiction that these sets cannot be equal. Suppose $V^1 \equiv V^2$. Let $P_{sv}^1 = \{u_0, (u_0, u_1), u_1, \ldots, (u_{k-1}, u_k), u_k\}$ and $P_{sv}^2 = \{w_0, (w_0, w_1), w_1, \ldots, (w_{k-1}, w_k), w_k\}$. Note $s = u_0 = w_0$ and $v = u_k = w_k$. Since $P_{sv}^1 \neq P_{sv}^2$, there is at least one index $0 < i < k$ for which $u_i \neq w_i$. Let $i$ be the smallest index for which $u_i \neq w_i$. From node $u_{i-1}$, both the labels of node $u_i$ and $w_i$ are updated. Since we suppose $V^1 \equiv V^2$, there is a node $w_j = u_i$ with $i < j < k$ and there is a node $u_h = w_i$ with $i < h < k$. Since $P_{sv}^1$ and $P_{sv}^2$ are equal up to index $i-1$, both $u_i$ and $w_i$ are labeled in the same iteration, having node $u_{i-1}$ as pivot node. For path $P_{sv}^1$, the following holds at some later iteration:

$$\sum_{n=0}^{i-1} c_{u_n, u_{n+1}} + \sum_{n=1}^{h-1} c_{w_n, w_{n+1}} < \sum_{n=0}^{i-1} c_{w_n, w_{n+1}}. \quad (3.2)$$
For path $P^2_{sv}$, the following holds at some later iteration:

$$
\sum_{n=0}^{i-1} c_{w_n,w_{n+1}} + \sum_{n=i}^{j-1} c_{w_n,w_{n+1}} < \sum_{n=0}^{i-1} c_{u_n,u_{n+1}}. 
$$  \hspace{1cm} (3.3)

Since all arc lengths are nonnegative, each sum in (3.2) and (3.3) is nonnegative. So (3.2) and (3.3) form a contradiction as $\sum_{n=0}^{i-1} c_{u_n,u_{n+1}} < \sum_{n=0}^{i-1} c_{w_n,w_{n+1}} < \sum_{n=0}^{i-1} c_{u_n,u_{n+1}}$. We conclude that $\mathcal{V}_1$ and $\mathcal{V}_2$ cannot be equal.

We now use this result to count all possible label updates. All possible paths belonging to a label update contain at most $|V| - 1$ arcs. Let $0 \leq a \leq |V| - 1$. The number of possible paths containing $a$ arcs, leading to a label update, is:

$$\binom{|V| - 1}{a}.$$

Since the first node is fixed (node $s$), we can choose $a$ extra nodes from $|V| - 1$ nodes in total. The maximum number of label updates is therefore:

$$\sum_{a=0}^{|V|-1} \binom{|V| - 1}{a} = 2^{|V|-1}.$$

Thus, the number of iterations is $O(2^{|V|})$. \hfill $\square$

In Appendix C, we extend this proof to graphs that contain negative arc costs, but no cycles of negative length (see Theorem C.6).

### 3.1.4 A special case: unit lengths

In case $c_{vw} = 1$ for all $(v,w) \in A$, the Generic Shortest Path algorithm has running time $O(|V||A|)$ (in accordance with Theorem 3.3). Moore [137] formulated an algorithm (Algorithm 3.5) to determine a shortest path tree in such a graph, which nowadays is called ‘breadth-first search’. In this algorithm, $\mathcal{V}_i$ denotes the set of nodes of $\mathcal{G}$ at distance $i$ from $s$.

**Algorithm 3.5: Breadth-First Search ($\mathcal{G}, s$)**

1: $i \leftarrow 0$
2: $\mathcal{V}_0 \leftarrow s$
3: repeat
4:   \( \mathcal{V}_{i+1} \leftarrow \{ w \in \mathcal{V} \setminus (\mathcal{V}_0 \cup \mathcal{V}_1 \cup \ldots \cup \mathcal{V}_{i-1}) | (v, w) \in \delta^+(\mathcal{V}_i) \} \)
5:   $i \leftarrow i + 1$
6: until $\mathcal{V}_i = \emptyset$
Algorithm 3.5 runs in $O(|V| + |A|)$ time, since every node and every arc will be explored once. Thus the shortest path tree rooted at node $s$ can be found in linear time if the graph has arcs of unit length only.

### 3.2 Label-setting algorithms

There are many implementations of the Generic Shortest Path algorithm. They mainly differ in how they select the pivot node $v \in Q$. Gilsinn and Witzgall [84] partitioned labeling algorithms into two general classes, *label-setting* and *label-correcting*. In this section, we will discuss label-setting algorithms.

In a label-setting algorithm, the selected pivot node $v \in Q$ has the minimum label compared to all other nodes in $Q$. The first formal label-setting algorithm was proposed by Dijkstra [59], and independently by Dantzig [40], and by Whiting and Hillier [198]. It is nowadays commonly referred to as Dijkstra, see Algorithm 3.6.

**Algorithm 3.6: Dijkstra ($G, s$)**

1. **initialize-single-source** ($G, s, Q$)
2. repeat
3. Select a pivot node $v \leftarrow \arg\min\{d_v | v \in Q\}$
4. $Q \leftarrow Q - \{v\}$
5. foreach $(v, w) \in \delta^+(v)$
6. \hspace{1em} **label-update** ($v, w, Q$)
7. until $Q = \emptyset$

Since Algorithm 3.6 is a specialization of Algorithm 3.4, Theorem 3.2 holds for Algorithm 3.6. Thus, **Dijkstra** is finite and correct for a strongly connected graph with nonnegative arc lengths.

**Theorem 3.5** [19] *For Dijkstra* (on a graph with nonnegative arc lengths) at any iteration, the following conditions hold for the set $R = \{v|d_v < \infty, v \notin Q\}$:

(i) No node belonging to $R$ at the start of the iteration will enter the candidate set $Q$ during the iteration.

(ii) At the end of the iteration, $d_u \leq d_w$ for all $u \in R$ and $w \notin R$.

**Proof**: We prove this by induction. At the beginning of the first iteration $R = \emptyset$, so (i) holds. At the end of the first iteration, we have $R = \{s\}$ and $d_s = 0$. Since all arc lengths are nonnegative, $d_s$ is a lower bound to all other labels and thus (ii) holds. Suppose that (i) and (ii) hold for an iteration. During the next iteration, node $v = \arg\min\{d_v | v \in Q\}$ is selected as pivot node. Only the labels of nodes $w$, $(v, w) \in \delta^+(v)$, with $d_w > d_v + \epsilon_{vw}$, decrease during the iteration and only those nodes enter $Q$ during the iteration. Let $w \in R$. By the induction hypothesis, we
3.2 Label-setting algorithms

have $d_w \leq d_v$. Since $c_{vw} \geq 0$, the label of node $w$ will not decrease during the iteration, following that node $w$ cannot enter $Q$. Furthermore, if the label of a node $w$ is decreased during the iteration, its new label is at least $d_v$. The only node added to $\mathcal{R}$ during the iteration, is node $v$. So for any node $u \in \mathcal{R}$ at the end of the iteration, we have $d_u \leq d_v \leq d_w$ for all $w \notin \mathcal{R}$. This completes the induction proof.

As a corollary of Theorem 3.5, each time a pivot node $v$ is selected, it enters the set $\mathcal{R}$ and so $v$ will not enter $Q$ anymore. Thus, label $d_v$ will not change from the moment where $v$ is selected as a pivot node. This is why Dijkstra belongs to the family of ‘label-setting algorithms’. During each iteration, the label of one node (the pivot node) becomes permanent. At that moment, the length of a shortest path from node $s$ to the pivot node is known.

### 3.2.1 Improved stop condition

In a label-setting algorithm, as soon as a pivot node $v$ is selected, the label $d_v$ is permanent and equals the length of a shortest $s$-$v$ path. Subsequently, we can improve the stop condition in Dijkstra (Algorithm 3.6). As soon as all nodes $t \in \mathcal{T}$ have a permanent label, we know the length of a shortest path from node $s$ to any node $t \in \mathcal{T}$. Thus, we can improve the stop condition in line 7 of Algorithm 3.6. We stop as soon as $\mathcal{T} \subset \mathcal{R}$.

At the moment $\mathcal{T} \subset \mathcal{R}$, the candidate set $\mathcal{Q}$ usually is not empty yet. The number of iterations is therefore usually less than the original stop condition. Two more remarks can be made:

- The stop condition $\mathcal{T} \subset \mathcal{R}$ can be checked directly after selecting a pivot node $v$. If the stop condition holds, it does not make sense to scan the outgoing arcs of node $v$.
- Selecting a pivot node $v = \arg\min\{d_v|v \in \mathcal{Q}\}$ means that it might happen that we can choose between multiple pivot nodes with the same label value. Ties should be broken in favor of a pivot node belonging to $\mathcal{T}$ [141].

### 3.2.2 Complexity

The candidate set $\mathcal{Q}$ in a label-setting algorithm is typically implemented as a so-called ‘priority queue’. Before we describe some common data structures that can be used to implement a priority queue, we first focus on a somewhat naive implementation, where $\mathcal{Q}$ is simply maintained as an unordered list.

**Proposition 3.6** [19] *The running time of Algorithm 3.6 is $O(|\mathcal{V}|^2 + |\mathcal{A}|)$.*

**Proof:** Procedure INITIALIZE-SINGLE-SOURCE requires $O(|\mathcal{V}|)$ operations. Each node $v$ enters $\mathcal{Q}$ at most once (see Theorem 3.5). Therefore, each node $v$ is selected
as a pivot node at most once. Thus, the number of iterations is $O(|V|)$. During each iteration, a node with the smallest label among all nodes in $Q$ has to be determined. Using a naive implementation of $Q$, this will take $O(|V|)$ operations. Furthermore, for each pivot node $v$, each arc $(v, w) \in \delta^+(v)$ has to be scanned and the label $d_w$ has to be updated if $d_v + c_{vw} < d_w$. Since each node $v$ is selected to be a pivot node at most once, the number of operations for scanning is bounded by $\sum_{v \in V} \delta^+(v) = |A|$. Thus, a label-setting algorithm has a running time $O(|V|^2 + |A|)$.

Since we assume the graph $G$ to be simple, we have $|A| < |V|^2$. If the graph represents a road map, typically $|A| = O(|V|)$. Thus, a label-setting algorithm, using a naive implementation for $Q$, has running time $O(|V|^2)$.

### 3.2.3 Data structures

Parnas [149] introduced several criteria for decomposing systems into modules. One idea is to define data structures by their operations only. Nowadays, this is called an abstract data type.

**Definition 3.7** An abstract data type (ADT) models a certain class of data structures with similar behavior. An ADT is defined indirectly, only by the operations that may be performed on it.

**Definition 3.8** A priority queue is an ADT that supports the following operations:

- **CREATE**. Return a new, empty priority queue.
- **INSERT($v$, $k$)**. Add an item $v$ with a predefined key $k$ to the priority queue.
- **FIND-MIN**. Return an item $v$ with the smallest key among all items in the priority queue.
- **DELETE-MIN($v$)**. Delete an item $v$ with the smallest key among all items in the priority queue and return $v$.
- **DECREASE-KEY($v$, $k$)**. Assign the new (lower) key $k$ to item $v$.

Sometimes (e.g. [133]) there are two more operations specified for priority queues: **UNION($Q_1$, $Q_2$)**, which returns the union of two priority queues $Q_1$ and $Q_2$, and **DELETE($v$)** to delete an arbitrary item from the priority queue. Since we do not need these operations, we omit them in Definition 3.8.

If we maintain $Q$ as a priority queue in *Dijkstra* (Algorithm 3.6), the operation **CREATE** is performed once, the operations **INSERT**, **DELETE-MIN** are performed $|V|$ times at most and **DECREASE-KEY** is performed at most $|A|$ times. Although we do not explicitly use **FIND-MIN** here, this operation is of interest to us, since we need it in several extensions of *Dijkstra*.

A well-known data structure that implements a priority queue is the **heap**. The term **heap** was introduced by Williams [199] in his **HeapSort** algorithm. That
3.2 Label-setting algorithms

Algorithm was an enhanced version of the TreeSort algorithm by Floyd [67]. Floyd in turn improved Williams’ algorithm and named that algorithm TreeSort3. [69]. Nowadays, the heap as introduced by Williams is called a ‘binary heap’. Johnson [114] generalized the binary heap to a so-called ‘d-heap’. A binary heap is equal to a 2-heap. If \( Q \) is implemented as a d-heap, the operations Insert, Delete-Min and Decrease-Key run in \( O(\log_d |Q|) \) time and the operations Create and Find-Min can be performed in \( O(1) \) time (see [35]). Using a d-heap to maintain \( Q \), Dijkstra runs in \( O(d|V| \log_d |V| + |A| \log_d |V|) \) time [115].

Over the years, heaps became more and more popular in a lot of applications. Several other heaps were invented. Some of them, like the binomial heap invented by Vuillemin [194] and improved by Brown [24], only provide a better worst case bound for the Union operation, which we do not need. We restrict our research to variants that are especially used in shortest path algorithms, like the Fibonacci heap [73] and the relaxed heap [62]. The Fibonacci heap performs the operations Create, Insert, Find-Min and Decrease-Key in \( O(1) \) amortized time (see Section 2.1.7) and the operation Delete-Min is performed in \( O(\log |Q|) \) amortized time. A relaxed heap achieves the same amortized bounds as a Fibonacci heap for each operation. There exists a variant of a relaxed heap for which those bounds are even worst-case bounds (instead of amortized bounds). Using a Fibonacci heap or a relaxed heap to maintain \( Q \), Dijkstra runs in \( O(|A| + |V| \log |V|) \) time.

A heap is surely not the only data structure that could be used to implement a priority queue. Dial [57] and Wagner [195] (independently, cf. [3]) introduced a bucket data structure that can be used on graphs with integral nonnegative length.

Let \( C \) be the maximum cost assigned to any arc:

\[
    C = \max_{(v,w) \in A} c_{vw}.
\]

For any node \( v \), with \( d_v < \infty \), we have \( 0 \leq d_v \leq C|V| \). In Dial’s data structure, for every possible value \( k \) of \( d_v \), a bucket \( k \) is maintained. This leads to \( C|V| + 1 \) buckets, numbered \( 0, 1, \ldots, C|V| \). In bucket \( k \) all nodes \( v \in Q \), with \( d_v = k \), are stored. Adding and removing a node from a bucket takes \( O(1) \) time. Thus, the operations Create, Insert and Decrease-Key take \( O(1) \) time as well. Since in a label-setting algorithm nodes are removed in a non-decreasing order of \( d_v \) (see Theorem 3.5), as soon as node \( v \) is selected as a pivot node, all buckets \( 0, 1, \ldots, d_v - 1 \) will be empty. To determine the next pivot node, we have to find the next non-empty bucket. Since we assume the graph to be strongly connected, from bucket \( d_v \) on (where \( v \) was the pivot node of the previous iteration) at most \( |C| - 1 \) buckets can be empty. The operation Delete-Min therefore takes \( O(C) \) time in the worst case. Dijkstra, using Dial’s data structure, runs in \( O(|A| + C|V|) \) time.

In Appendix B, we will discuss the abovementioned as well as some other data structures that implement a priority queue, in more detail.
3.3 Label-correcting algorithms

Recall that in a label-setting algorithm, node $v$ with the smallest label $d_v$ in $Q$ is chosen to be the pivot node. Finding that node $v \in Q$, where $d_v$ is actually as small as possible, is quite time consuming. To avoid this, one may consider other nodes to be the pivot node of an iteration. This is done by the so-called label-correcting algorithms. These algorithms maintain a queue $Q$ and the pivot node is always the first node of that queue. Thus, the selection of the node to be removed from the candidate list $Q$ is simpler and requires less overhead than in label-setting algorithms, at the expense of multiple entrances of nodes in $Q$. All these algorithms use some type of queue to maintain the candidate list $Q$. They differ in the way the queue is structured and where nodes are inserted in the queue.

When a node is first removed from $Q$, its label value does not necessarily contain the shortest path from $s$ to this node, as is the case with label-setting algorithms. The labels may improve and re-enter $Q$. No shortest path is known until the end of the algorithm, where $Q = \emptyset$ and a shortest path to any node $t \in T$ is determined.

3.3.1 Potential-invariant algorithms

Before we discuss some of the earliest label-correcting algorithms, we present some theory based on [29].

**Definition 3.9** A potential function $f : V \to \mathbb{R}$ is a real-valued function on nodes. Given a potential function $f$, we define the reduced arc costs as $c_f(v, w) = c_{vw} + f_v - f_w$. If both $f_v = \infty$ and $f_w = \infty$, we define $c_f(v, w) = c_{vw}$.

The node labels $d$ are real-valued and so $d$ can be seen as a potential function.

**Definition 3.10** Let $f$ be a potential function and let $G = (V, A)$ be a digraph. An arc $(v, w) \in A$ is called admissible if $c_f(v, w) \leq 0$. The set of all admissible arcs is denoted by $A_f$ and the admissible graph is defined as $G_f = (V, A_f)$.

It is easy to see that arc $(v, w) \in A$, with $f_v < \infty$ and $f_w = \infty$, is admissible.

**Proposition 3.11** [29] Let $f$ be a potential function and let $G = (V, A)$ be a digraph with positive arc costs. Then the admissible graph $G_f$ is acyclic.

**Proof:** We prove this by contradiction. Suppose $G_f = (V, A_f)$ contains a cycle $(v_0, a_1, v_1, \ldots, a_k, v_k)$ where $v_k = v_0$. Since $a_i \in A_f$ for $1 \leq i \leq k$, we have: $c_{v_{i-1}, v_i} + f_{v_{i-1}} - f_{v_i} \leq 0$ for $1 \leq i \leq k$. Addition of these inequalities shows:

$$
\sum_{i=1}^{k} (c_{v_{i-1}, v_i} + f_{v_{i-1}} - f_{v_i}) = \sum_{i=1}^{k} c_{v_{i-1}, v_i} \leq 0.
$$

(3.4)
Since all arc costs are positive, the length of any cycle is positive. This is in contradiction with (3.4).

It should be noted that if $G = (V, A)$ has nonnegative arc costs, it is possible that $G$ contains a cycle with zero length. In such a case, $G_f$ might contain a zero length cycle too.

**Definition 3.12** Two instances of the shortest path problem are called equivalent if the underlying graphs $G' = (V, A)$ and $G'' = (V, A)$ are identical and the arc costs $c''_{vw} = c'_f(v, w)$ for some potential function $f$. Since $G'$ and $G''$ are identical, their representations (ordering of the nodes and the ordering of the outgoing arcs of each node) are identical.

**Proposition 3.13** [63] Let $G' = (V, A)$ be a graph with arc costs $c'_{vw}$ and let $G'' = (V, A)$ be a graph with arc costs $c''_{vw} = c'_f(v, w)$ for some potential function $f$. For each pair of nodes $v, w \in V$, a $P_{vw}$ path is a shortest path in $G'$ if and only if it is a shortest path in $G''$.

**Proof:** Clearly, the length of some path $P$ in $G''$ between two nodes $v$ and $w$ equals:

$$\ell(P) := \sum_{(v, w) \in P} c''_{vw} = f_v - f_w + \sum_{(v, w) \in P} c'_vw.$$ 

Since $f_v - f_w$ is constant, any shortest $v$-$w$ path in $G''$ is also shortest in $G'$ and vice versa.

**Definition 3.14** A label-correcting algorithm is called potential-invariant if it performs the same sequence of node scans on two equivalent problem instances.

Since the label-correcting algorithms in this section use representations of $Q$, in which the values of the labels in the queue have no influence on the order of the nodes in $Q$, these algorithms are potential-invariant.

### 3.3.1.1 Bellman-Ford-Moore (queue)

This is one of the simplest label-correcting algorithms. It is called the **Bellman-Ford-Moore** algorithm, since it is closely related to an algorithm proposed by Bellman [14, 15], Ford [70] and Moore [137].

A node is always removed from the top of the queue and inserted at the bottom of the queue; a first-in, first-out (FIFO) rule is used to update the queue.

**Proposition 3.15** [147] The Bellman-Ford-Moore algorithm can be rewritten in the form of Algorithm 3.7.
Algorithm 3.7: Bellman-Ford-Moore [35]

1: INITIALIZER-SINGLE-SOURCE($G$, $s$, $Q$)
2: for $i \leftarrow 1$ to $|V| - 1$ do
3:   foreach $(v, w) \in A$ do
4:     LABEL-UPDATE($v$, $w$, $Q$)

Proof: The iterative process of Algorithm 3.7 can be seen as the dynamic programming relation:

$$d^0_s = 0, \; d^0_v = \infty \; \forall \; v \in V \setminus \{s\},$$

$$d^{k+1}_w = \min(d^k_w, \min\{d^k_v + c_{vw} | (v, w) \in \delta^-(w)\}).$$

(3.5) (3.6)

The dynamic programming relation (3.6) is the original solution method from [70]. In this relation, $d^k_w$ is the length of a shortest path from $s$ to $w$, containing at most $k$ arcs. It is obvious that each simple path includes less than $|V|$ arcs. So we know that $d^{|V|}_w = d_w$ for all $w \in V$.

To eliminate unnecessary operations, a priority queue $Q$ can be introduced:

$$Q^0 = \{s\},$$

$$Q^k = \{v \in V | d^k_v < d^{k-1}_v\}. \quad (3.7)$$

We rewrite (3.6), using (3.7):

$$d^{k+1}_w = \min(d^k_w, \min\{d^k_v + c_{vw} | (v, w) \in \delta^-(w), v \in Q^k\}). \quad (3.8)$$

Ordering the operations of (3.8) differently, we find:

$$d^{k+1}_w = d^k_w, \; w \in V,$$

$$d^{k+1}_w = \min(d^{k+1}_w, \min\{d^k_v + c_{vw} | (v, w) \in \delta^-(w), v \in Q^k\}). \quad (3.9)$$

The Generic Shortest Path algorithm, where the candidate list $Q$ is implemented as a queue, has the same behavior as solving (3.9). The Generic Shortest Path algorithm starts with $Q = \{s\}$, like $Q^0$ in (3.7). Determining one cycle $k + 1$ in (3.9) is similar to performing all iterations of the Generic Shortest path algorithm for the nodes in $Q$ at that moment. If a new node has to be added to $Q$ it is added to a new queue $Q'$ (instead of directly to $Q$). At the end of the cycle (so $Q = \emptyset$), we set $Q \leftarrow Q'$.

Proposition 3.16 The Bellman-Ford-Moore algorithm requires $O(|V||A|)$ operations. The Generic Shortest Path algorithm, with $Q$ implemented as a queue, performs $|V|^2$ iterations.
Proof: Consider Algorithm 3.7. The initialization in line 1 takes \( O(|V|) \) time, each of the \(|V| - 1\) passes over the edges in lines 2–4, takes \( O(|A|) \) time. Thus, the Bellman-Ford-Moore algorithm runs in \( O(|V||A|) \) time.

Since \( d_w^{[\nu]} = d_w \) as seen above, we may state that a node \( w \) re-enters \( Q \) at most \(|V|\) times in the Generic Shortest Path algorithm where \( Q \) is implemented as a queue. Therefore, the number of iterations is at most \(|V|^2\).

Despite the generally larger number of iterations required by Bellman-Ford-Moore over Dijkstra, Bellman-Ford-Moore can be superior in practice because of the smaller overhead per iteration [97].

Remark 3.1 For acyclic graphs, the number of iterations of the Bellman-Ford-Moore is exactly \(|V|\), and similar to Dijkstra. However, a road map is not acyclic.

3.3.1.2 Gilsinn and Witzgall (stack)

Gilsinn and Witzgall [84] presented an algorithm in which the candidate set \( Q \) is implemented as a stack (last-in, first-out), resulting in a depth-first exploration of the graph.

Proposition 3.17 [179] The algorithm of Gilsinn and Witzgall requires \( \Theta(2^{\nu^1}) \) iterations.

Proof: From Theorem 3.4, the number of iterations of the Generic Shortest Path algorithm is \( O(2^{\nu^1}) \). To prove this bound is tight, we give an example where this number of iterations actually occurs. Consider the family of graphs \( \mathcal{G}^n = (\mathcal{V}^n, \mathcal{A}^n) \) as shown in Figure 3.1, with \(|\mathcal{V}^n| = 2n + 1\) and \(|\mathcal{A}| = 3n\). Suppose the list of arcs adjacent to node \( v \in \mathcal{V}^n \) are sorted in ascending order of the arc costs. Let \( E(n) \) be the number of times the LABEL-UPDATE procedure is executed while determining the shortest path from node 0 to node \( 2n \) in graph \( \mathcal{G}^n \). We prove by induction that:

\[
\begin{align*}
E(0) &= 0, \\
E(n + 1) &= 3 + 2E(n), \\
d(0, 2n) &= 2^n - 1.
\end{align*}
\]

Since \( \mathcal{G}^0 \) contains only one node and no arcs, clearly \( E(0) = 0 \) and \( d(0, 0) = 0 \). \( \mathcal{G}^1 \) contains 3 nodes and 3 arcs. All arcs have to be scanned to determine the shortest path from node 0 to node 2, so \( E(1) = 3 + 3 + E(0) \). It is easy to see that \( d(0, 2) = 1 \). Now suppose \( E(n) = 3 + 2E(n - 1) \) and \( d(0, 2n) = 2^n - 1 \) for a certain \( n \geq 1 \). Graph \( \mathcal{G}^{n+1} \) is an extension of graph \( \mathcal{G}^n \). Nodes 0, 1, \ldots, \( 2n \) of graph \( \mathcal{G}^n \) are numbered 2, 3, \ldots, \( 2(n + 1) \) in graph \( \mathcal{G}^{n+1} \). There are two new nodes \((0, 1)\) and three new arcs \(((0, 1), (0, 2)\) and \((1, 2))\) in graph \( \mathcal{G}^{n+1} \), compared to graph \( \mathcal{G}^n \). At the beginning of the algorithm, node 0 is selected from the stack \( Q \). During the first iteration, the outgoing arcs of node 0 are scanned and nodes 1 and 2 are pushed onto the stack.
Different methods for solving the SPP

The next pivot node is node 2, having the property \( d_2 = 2 \) \((n+1)\). Using the induction hypothesis, \( 3 + E(n-1) \) label updates have to be performed before node 1 is the node on top of the stack \( \mathcal{Q} \). Next, node 1 is chosen as pivot node. The label of node 2 can be decreased from node 1 to \( d_2 = d_1 + 0 = 2^n \), by scanning arc \((1, 2)\). Again, by the induction hypothesis, \( E(n) = 3 + E(n-1) \) label updates have to be performed before the stack \( \mathcal{Q} \) is empty. The total amount of label updates is then \( E(n+1) = 3 + 2(3 + E(n-1)) = 3 + 2E(n) \). Furthermore, \( d(0, 2(n+1)) = d_2 + d(2, 2(n+1)) = 2^n + 2^n - 1 = 2^{(n+1)} - 1 \). This completes the induction proof.

From (3.10), it follows that the total number of label updates equals \( 3(2^n - 1) \). Since \(|\mathcal{V}^n| = 2n\), the algorithm of Gilsinn and Witzgall requires \( O(2^{|\mathcal{V}|}) \) label updates. A node is added to the stack in each label update. Thus, the number of iterations is \( \Theta(2^{|\mathcal{V}|}) \).

---

**Figure 3.1:** A family of graphs

3.3.1.3 D’Esopo-Pape (stack + queue)

In 1974, Pape [148] exploited a suggestion by D’Esopo (as stated in [160]) and proposed a label-correcting algorithm in which a node is always removed from the top of the queue. A node, upon entrance in the queue, is placed at the bottom of the queue if it has never been in the queue before; otherwise it is placed at the top of the queue. The candidate set \( \mathcal{Q} \) is implemented as a union of a stack \( \mathcal{Q}' \) and a queue \( \mathcal{Q}'' \), as shown in Figure 3.2. This algorithm is nowadays known as D’Esopo-Pape.

The idea is that, when a node \( v \) is removed from \( \mathcal{Q} \), its label affects the labels of a subset \( \mathcal{B}_v \) of nodes in the direct neighbourhood of \( v \). When the label of \( v \) changes again, it is likely that the labels of the nodes in \( \mathcal{B}_v \) will require updating as well. It
is thus argued that it makes sense to place the node at the top of the queue so that the labels of the nodes in $B_v$ get a chance to be updated as quickly as possible.

**Proposition 3.18** [120, 179] The number of iterations required by D’Esopo-Pape is $\Theta(2^{|V|})$.

**Proof:** From Theorem 3.4, the number of iterations of the Generic Shortest Path algorithm is $O(2^{|V|})$. To prove this bound is tight, we give an example where this number of iterations actually occurs. Consider the graph in Figure 3.3 and let node 1 be the origin. Suppose that the list of outgoing arcs from node 1 (the origin) is sorted in descending order of arc costs, and that the list of outgoing arcs of all other nodes is sorted in ascending order of arc costs. Every time a node other than node 2 is selected as a pivot node, the label of node 2 is updated and node 2 is the pivot node in the next iteration. In fact, node 2 takes all (integer) labels between 20 and 5 and is added to $Q$ each time its label is updated. Here, $|V| = 6$ and node 2 enters the candidate set $2^{|V| - 2}$ times. Graphs with this characteristic and of arbitrary size can be formed using Algorithm 3.8. On such a graph, during the execution of D’Esopo-Pape, node 2 takes all labels from $|V| - 1$ to $2^{|V| - 2} + |V| - 2$ and thus enters the candidate set $2^{|V| - 2}$ times.

![Graph](image)

**Figure 3.3:** A pathological graph where $|V| = 6$

It should be noted that this pathological situation is a case of not only the unusual arc lengths, but also of the ordering of the lists of outgoing arcs. If those lists were ordered oppositely, each node would be scanned only once.

It is not surprising that D’Esopo-Pape needs $\Theta(2^{|V|})$ iterations [179]. D’Esopo-Pape actually becomes the algorithm of Gilsinn and Witzgall once all nodes have entered the candidate set $Q$. For example, the family of graphs $G^n$ as shown in Figure 3.1 can be modified by adding arcs from the origin 0 to any node $v$, $3 \leq v \leq 2n$. The arc costs of these arcs are set to a large constant $M > 2^{n+1}$. Furthermore,
Algorithm 3.8: A pathological graph [120]

1: add node 1 and node 2
2: add an arc (1, 2) with $c_{12} \leftarrow 1$
3: add an arc (2, 1) with $c_{21} \leftarrow 2$
4: for $k \leftarrow 3$ to $|V|$ do
5:   add node $k$
6:   for $i \leftarrow 2$ to $k - 1$ do
7:       add an arc $(k, i)$ with $c_{ki} \leftarrow c_{1i}$
8:       set $c_{1i} \leftarrow c_{1i} + 2^{|V| - 3} + 1$
9:   add an arc $(1, k)$ with $c_{1k} \leftarrow 1$

the list of adjacent arcs of node 0 are now sorted in a non-increasing order of arc costs. The list of adjacent arcs of all other nodes is still ordered according to non-decreasing arc costs. After the first iteration, all nodes are in $Q$. Subsequently, all nodes $v$ with $d_v = M$ will act as a pivot node but no label will be updated during these iterations. From that moment on, the candidate set $Q$ contains nodes 2 and node 1 (in that order). From this point, D’ESOPO-PAPE is equivalent to Gilsinn-Witzgall when applied to the original graph $G_n$.

Despite the fact that the number of iterations is $\Theta(2^{|V|})$, D’ESOPO-PAPE performs very well in practice. For sparse graphs, it usually outperforms BELLMAN-FORD-MOORE and it is competitive with the best label-setting methods [18]. No definitive explanation has yet been given for this success.

As we saw earlier, there is a strong dependence of the performance of the algorithm on a pre-ordering of the outgoing arcs of each node. The general idea would be to order the outgoing arcs of each node in such a way that the neighbors of that node are likely to be scanned in ascending order of their respective labels.

3.3.1.4 Pallottino (two queues)

Pallottino [146] improved D’ESOPO-PAPE by splitting the candidate set $Q$ into two distinct queues $Q'$ and $Q''$ (see Figure 3.4). If a node has never before been in any of these two queues, it is inserted at the bottom of $Q''$. Otherwise, it is inserted at the bottom of $Q'$. The pivot node $v$ leaving the candidate set in any iteration, is the top node of $Q'$ if $Q' \neq \emptyset$, otherwise it is the top node of $Q''$. We call the resulting algorithm after the author PALLOTTINO.

Proposition 3.19 [147] The PALLOTTINO algorithm runs in $O(|V|^2 |A|)$ time.

Proof: PALLOTTINO (as many other algorithms, including D’ESOPO-PAPE) can be seen as a graph growth algorithm. A graph growth of $\mathcal{G}$ is defined to be any subset
of graphs $G^1, G^2, \ldots, G^N$, such that
\begin{equation}
G^n \subseteq G^{n+1}, \quad n = 1, 2, \ldots, N - 1, \\
G^N \equiv G.
\end{equation}

Let $p(\mathcal{V})$ be a permutation of the nodes in $\mathcal{V}$; Let $\mathcal{V}^n$ be the set of the first $n$ nodes of the permutation $p(\mathcal{V})$. If the $n^{th}$ graph of the growth, $G^n$, is such that:
\begin{equation}
\mathcal{A}^n = \{(v, w) | v \in \mathcal{V}^n\}, \quad G^n = (\mathcal{V}, \mathcal{A}^n),
\end{equation}
then the growth is called a \textit{graph growth of} $G$ \textit{by partial graphs}.

Let $p(\mathcal{V})$ be the permutation corresponding to the order in which nodes are selected to be a pivot node for the first time (the moment where a node leaves $Q''$). Thus $\mathcal{V}^n$, the set of the first $n$ nodes of the permutation $p(\mathcal{V})$, is the set of all nodes that were removed at least once from $Q$.

Finding a shortest path tree in $G$ is equivalent to finding a shortest path tree in the $|\mathcal{V}|$ graphs of the graph growth (3.12) for the permutation $p(\mathcal{V})$. In iteration $n, 1 \leq n \leq |\mathcal{V}|$ the shortest path tree of $G^n$ is determined. These iterations are called \textit{master} iterations, to distinguish them from the iterations of the base algorithm (in this case PALLOTTINO), that we will call \textit{base} iterations. Note that we only execute base iterations (running the base algorithm) and while doing so, only trigger master iterations.

Let $v^n$ be the $n^{th}$ node removed from $Q''$, starting the $n^{th}$ master iteration. During the execution of this master iteration, only nodes $v \in \mathcal{V}^n$ are added to $Q'$. As soon as $Q' = \emptyset$, the master iteration ends and the shortest path tree of $G^n = (\mathcal{V}, \mathcal{A}^n)$ is found. Note that $\mathcal{A}^n = \mathcal{A}^{n-1} \cup \delta^+(v^n)$.

The complexity of PALLOTTINO consists of two elements: the complexity $b(|\mathcal{V}|, |\mathcal{A}^n|)$ of the base algorithm. This transforms the shortest path tree of $G^{n-1}$ into the shortest path tree of $G^n$, summed over the different main iterations. The second element is the initialization step. The resulting complexity is:
\begin{equation}
O \left( |\mathcal{V}| + \sum_{n=1}^{|\mathcal{V}|} b(|\mathcal{V}|, |\mathcal{A}^n|) \right).
\end{equation}
In PALLOTTINO, $Q'$ is implemented as a queue. So $b(|V|, |A^u|) = O(|V| |A|)$, as shown in Proposition 3.16. Therefore, the complexity of PALLOTTINO is $O(|V|^2 |A|)$.

In practice, D’ESOPO-PAPE and PALLOTTINO behave quite similarly [76, 121]. Since PALLOTTINO (at least in their experiments) has always proved to be (almost) as good as D’ESOPO-PAPE, without the risk of bad behavior in pathological cases, it can be recommended over D’ESOPO-PAPE.

3.3.1.5 The Partitioning Shortest Path algorithm

Glover, Klingman and Phillips [87] introduced the Partitioning Shortest Path algorithm. The list $Q$ is partitioned into two disjoint sets $Q'$ and $Q''$. If a node has to be added to $Q$, it is always added to $Q''$. The pivot node is always selected as the first node in $Q'$. It might happen that $Q' = \emptyset$, while $Q'' \neq \emptyset$. Thus, if $Q' = \emptyset$ at the start of Line 3 of the Generic Shortest Path algorithm (Algorithm 3.4), the list $Q$ is repartitioned by transferring all nodes from $Q''$ to $Q'$ just before the pivot node is selected.

**Proposition 3.20** On a digraph $G = (V, A)$ with nonnegative arc costs, the Partitioning Shortest Path algorithm runs in $O(|V| |A|)$ time.

**Proof:** As soon as $Q$ is about to be repartitioned for the $k^{th}$ time, each node in $Q''$ has at least $k - 1$ predecessors as we will prove by induction: As soon as $Q$ is repartitioned for the first time (transferring node $s$ from $Q''$ to $Q'$), node $s$ clearly has no predecessors. Now suppose that, at the beginning of the $k^{th}$ repartitioning of $Q$, each node in $Q''$ has at least $k - 1$ predecessors. At that moment, $Q''$ is transferred to $Q'$. Let $R$ be the set of nodes in $Q'$ at that moment. As soon as all nodes in $Q'$ are scanned, $Q' = \emptyset$ and a new repartitioning action is needed. All nodes in $Q''$ now have an immediate predecessor in $R$. It might even happen that such a predecessor $v$ itself has an immediate predecessor in $R$, since $d_v$ can be updated while scanning all the nodes of $R$. Consequently, all nodes in $Q''$ now have at least $k$ predecessors. This completes the induction proof.

A shortest path has at most $|V| - 1$ predecessors. Thus, the set $Q$ cannot be repartitioned more than $|V|$ times. After repartitioning $Q$, any node $v$ with the smallest label among all nodes in $Q'$ will not appear in $Q''$ anymore (since all arc costs are nonnegative). Between two successive repartitionings of $Q$, each arc is examined only once, at most. Hence, the Partitioning Shortest Path algorithm has running time $O(|V| |A|)$.

The Partitioning Shortest Path algorithm is a generic algorithm, due to the unspecified way in which nodes are added to $Q''$ and $Q'$, both after updating a node label, as well as during the repartitioning of $Q$. For example, if $Q'$ is organized as a stack and $Q''$ is organized as a queue, the algorithm is still polynomially bounded.
This can be seen as a ‘cousin’ of the Gillis and Witzgall algorithm. Similarly, we can construct a version of the Partitioning Shortest Path algorithm that emulates the concept of D’Esopo-Pape: Let $Q''$ be organized as a stack and queue (like in D’Esopo-Pape), and $Q'$ as a queue.

In [87], it is shown that by changing the repartitioning procedure, the resulting Partitioning Shortest Path algorithm runs in $O(|V|^2)$ time. Let $n \in \mathbb{N}$ ($n \geq 1$) be a given constant, independent from $|V|$. If $|Q''| \leq n$, all nodes are transferred to $Q'$. Otherwise, only the first $n-1$ nodes are transferred from $Q''$ to $Q'$. Finally, a node $v \in Q''$, having the smallest label among all remaining nodes in $Q''$, is transferred to $Q'$. This transferring procedure ensures that $|Q'| \leq n$ and that $Q'$ contains a node with the minimum label among all nodes in $Q$. Thus, at least one node of $Q'$ will never re-enter $Q''$, since all arc costs are nonnegative. The number of repartitionings of $Q$ is thus bounded by $|V|$. After a repartitioning, $|Q'| \leq n$ and for any node $v \in Q'$, $|\delta^+(v)| \leq |V| - 1$. Thus, the amount of work between two repartitionings is bounded by $O(n|V|) = O(|V|)$. The resulting Partitioning Shortest Path algorithm has running time $O(|V|^2)$.

This version of the Partitioning Shortest Path algorithm uses the actual node label values in the repartitioning procedure. Therefore, this version is not a potential-invariant one, like the original version is. In the next section, we will describe several more label-correcting algorithms that are not potential-invariant.

### 3.3.2 Advanced label-correcting algorithms

The label-correcting algorithms presented in the previous section do not use the actual label values while maintaining the candidate set $Q$. In this section, we will mention some label-correcting algorithms that try to keep the candidate set $Q$ a little bit in order (in the sense that nodes with small label values are preferably scanned earlier than others).

These algorithms are based on the hypothesis that queue management strategies should try to place nodes with small labels near the top of the queue, thus approximately applying the minimum label selection policy of label-setting algorithms with a much smaller computational overhead. The more all nodes are stored in the queue by ascending label order, the less iterations are required. Note that for a node $w$ to re-enter $Q$, some node $v$ with $d_v + c_{vw} < d_w$ must first exit $Q$. Thus, the smaller $d_w$ was at the previous exit of $w$ from $Q$, the less likely it is that $d_v + c_{vw}$ will subsequently become smaller than $d_w$ for some node $v \in Q$ and arc $(v, w)$.

#### 3.3.2.1 The family of Threshold algorithms

Glover et al. [85] described a family of algorithms, the so-called Threshold algorithms, for solving the shortest path algorithm. A Threshold algorithm actually is a specification of the Generic Shortest Path algorithm (Algorithm 3.4) and closely related to the Partitioning Shortest Path algorithm (see Section 3.3.1.5). In
a \textbf{Threshold} algorithm, the candidate set $Q$ is divided into three subsets: $Q_1'$, $Q_2'$, and $Q''$. Compared to the \textbf{Partitioning Shortest Path} algorithm, the subset $Q'$ is now defined by $Q_1' \cup Q_2'$. Furthermore, a threshold parameter $\tau$ is maintained. Initially, $\tau = 1$, $Q_1' = Q_2' = \emptyset$ and $Q'' = \{s\}$.

Suppose that a node label $d_v$ is updated during the execution of the Generic Shortest Path algorithm. If $d_v > \tau$ and $v \notin Q$, then $v$ is added to $Q''$. If, on the other hand, $d_v \leq \tau$ and $v \notin Q_1' \cup Q_2'$, node $v$ is added to $Q_2'$ and removed from $Q''$ (if $v \in Q''$). The pivot node is always selected from $Q_1'$. If $Q_1' = \emptyset$ and $Q_2' \neq \emptyset$, $Q$ is repartitioned by transferring all nodes from $Q_2'$ to $Q_1'$, which can simply be achieved by exchanging the names $Q_1'$ and $Q_2'$. If $Q_1' = Q_2' = \emptyset$, the candidate set $Q$ is repartitioned in two steps:

1. The threshold parameter $\tau$ is updated, such that $\tau \leq \min\{d_v | v \in Q''\}$.
2. All nodes $v \in Q''$, for which $d_v \leq \tau$, are transferred to $Q_1'$.

Before we discuss the \textbf{Threshold} algorithms in more detail, let us focus on two degenerate versions of the algorithm.

- Suppose $\tau$ is always adjusted to $\tau \leftarrow \min\{d_v | v \in Q''\}$. The resulting \textbf{Threshold} algorithm is equivalent to \textbf{Dijkstra}.
- Let both $Q_1'$ and $Q_2'$ be implemented as a queue and suppose $\tau$ is updated to infinity. The resulting algorithm is equivalent to \textbf{Bellman-Ford-Moore}.

\textbf{Proposition 3.21} \cite{87} A \textbf{Threshold} algorithm runs in $O(|V||A|)$ time on a digraph $G = (V, A)$ with nonnegative arc costs.

\textbf{Proof}: Every time $Q$ is repartitioned (either by transferring nodes from $Q_2'$ to $Q_1'$ or by transferring nodes from $Q''$ to $Q_1'$), the label of node $v$ with $d_v = \min_{w \in Q} d_w$ cannot be improved anymore, due to the nonnegative arc costs. These nodes will never be added to $Q$ anymore. Thus, the number of repartitionings of $Q$ is bounded by $O(|V|)$. Between two successive repartitionings of $Q$, each arc is examined only once, at most. Hence, a \textbf{Threshold} algorithm has running time $O(|V||A|)$. \hfill $\square$

Similar to the \textbf{Partitioning Shortest Path} problem, this bound can be improved to $O(|V|^2)$, by transferring at most $n \in \mathbb{N}$ nodes at each repartitioning, where $n \geq 1$ and $n$ is independent of $|V|$. Recall that at least one node $v$ with $d_v = \min_{w \in Q} d_w$ should be transferred. At least one node of $Q_1'$ will never re-enter $Q$, since all arc costs are nonnegative. Thus, the number of repartitionings of $Q$ is still bounded by $|V|$. After a repartitioning, $|Q_1'| \leq n$ and for any node $v \in Q_1'$, $|\delta^+(v)| \leq |V| - 1$. Thus, the amount of work between two repartitionings is bounded by $O(n|V|) = O(|V|)$. The resulting \textbf{Threshold} algorithm has running time $O(|V|^2)$.

It should be noted that it is important to use the three sets $Q_1'$, $Q_2'$ and $Q''$ to achieve the worst-case bound of Proposition 3.21. In several articles \cite{18, 76} a \textbf{Threshold} algorithm is suggested where $Q$ is partitioned into only two sets $Q'$ and $Q''$, both
implemented as a queue. If node $v$ has to be added to $Q$, it will be added to the tail of $Q'$ if $d_v \leq \tau$ and to the tail of $Q''$ otherwise. This variant of the Threshold algorithm runs in $O(|V|^2|A|)$ time. A proof can be found in [76].

In [88], two methods are developed for updating the threshold parameter $\tau$, resulting in two members of the family of Threshold algorithms: Thresh-X1 and Thresh-X2. Both algorithms are reported to perform extremely well in practice, especially Thresh-X2. This behavior is verified by other experimental evaluations ([21, 111, 121, 135]).

A disadvantage of Thresh-X2 is that the update method of $\tau$ includes a domain-specific parameter. In [18], experimental evaluations for different parameter values are reported. The performance of Thresh-X2 is highly sensitive for the choice of this parameter. Thus, we may expect that in some experiments, Thresh-X2 is outperformed by other algorithms. In [205], Pallottino is recommended above Thresh-X2, while in [29], both Dijkstra and Pallottino are recommended instead of Thresh-X2.

### 3.3.2.2 Topological Scan algorithm

Goldberg and Radzik [93] present a Topological Scan algorithm that can be seen as a variant of the Partitioning Shortest Path problem [153]. The Topological Scan algorithm exploits the fact that if the reduced arc costs $c_d(v,w) < 0$ (see Definition 3.9), it is better to scan node $v$ before node $w$, since by scanning node $v$ the label $d_w$ will be improved.

Before we describe the algorithm in more detail, we present some definitions and notation according to [123].

**Definition 3.22** A partial ordering of a set $V$ is a relation between the elements of $V$, denoted by $\preceq$, satisfying the following properties for any $u, v, w \in V$:

(i) If $u \preceq v$ and $v \preceq w$, then $u \preceq w$.

(ii) If $v \preceq w$ and $w \preceq v$, then $v = w$.

(iii) $v \preceq v$.

If $v \preceq w$, we say that $v$ precedes or equals $w$.

**Definition 3.23** The problem of topological sorting is to embed a partial ordering of $V$ in a linear ordering. This arranges the items $v_i \in V$ into a linear sequence $v_1, v_2, \ldots, v_{|V|}$ such that, whenever $v_j \preceq v_k$, we have $j \leq k$.

Let $G = (V, A)$ be a digraph with positive arc costs. The admissible graph $G_d = (V, A_d)$ induced by $d$ (see Definition 3.10) is used to define a partial ordering: if $(v, w) \in A_d$, we write $v \preceq w$. Part (ii) of Definition 3.22 requires $G_d$ to be acyclic. Since the arc costs are positive, $G_d$ is indeed acyclic (see Proposition 3.11).
Like in the Partitioning Shortest Path problem, the candidate set $Q$ is partitioned into $Q'$ and $Q''$. Here, $Q'$ is a linearly ordered set. The pivot node is selected from $Q'$ according to the ordering of $Q'$. Nodes are added to $Q''$. If node $Q' = \emptyset$ and $Q'' \neq \emptyset$, the candidate set $Q$ is rebuilt by the following steps:

(i) Remove any node $v \in Q''$ if there is no arc $(v, w) \in \delta^+(v)$ for which $c_{d}(v, w) < 0$.

(ii) Let $Q'$ be the set of reachable nodes from any node in $Q''$ in the admissible graph $G_d$.\(^1\)

(iii) Topologically sort $Q'$ to embed the partial order induced by $A_d$ in a linear order (see Definition 3.23).

(iv) Set $Q'' = \emptyset$.

Knuth [123] provides the first algorithm for topological sorting that runs in $\Theta(|V| + |A_d|)$ time. Commonly, a Depth First Search is executed on $G_d$ to determine the topological order of the nodes in $Q'$ (see [35]). If $G = (V, A)$ contains zero length cycles, the graph $G_d$ is not necessarily acyclic. In case $G_d$ is not acyclic, it is suggested to simply ignore the so-called ‘back arcs’ while performing the Depth First Search [29, 93].

**Proposition 3.24** Let $G = (V, A)$ be a digraph with positive arc costs. The Topological Scan algorithm is correct.

**Proof:** Although we can easily embed the Topological Scan algorithm in the Partitioning Shortest Path algorithm, and thus in the Generic Shortest Path algorithm, to prove its correctness, there are two changes, performed at the time of repartitioning $Q$, for which it should be checked whether or not they influence the finiteness and correctness of the algorithm:

- The removing of node $v \in Q''$ if there is no arc $(v, w) \in \delta^+(v)$ for which $c_{d}(v, w) < 0$.
- The addition of some extra nodes to $Q'$, which are reachable from $Q''$ (in $G_d$), but not in $Q''$.

Removing any node $v \in Q''$, if there is no arc $(v, w) \in \delta^+(v)$ for which $c_{d}(v, w) < 0$, clearly does not affect the finiteness nor the correctness of the algorithm. Suppose node $v$ was added to $Q'$. At the time node $v$ is selected as a pivot node, either $d_v$ is equal to the label directly after the repartitioning took place, or $d_v$ is improved. If $d_v$ is still unchanged, we have $d_v + c_{vw} > d_w$, for all $(v, w) \in \delta^+(v)$. Based on $d_v$, no other node label will be updated. If $d_v$ is updated after the repartitioning,

\(^1\)In [93] it is assumed that all labels in the newly formed set $Q'$ have a finite label (in the proof of correctness of the algorithm), but it is not explicitly stated. If, for $(v, w) \in A$, we have $d_v < \infty$ and $d_w = \infty$, it follows by Definition 3.10 that $(v, w) \in A_d$. Thus, node $w$ is reachable from $v$. In our proof of correctness of the algorithm, we will not use the finiteness of the node labels.
node $v$ is added to $Q''$ and thus might be considered as a pivot node after the next repartitioning of $Q$.

Next, we prove that adding some extra nodes to $Q$ during the repartitioning of $Q$ does not influence the finiteness of the algorithm. The Generic Shortest Path algorithm terminates as soon as $Q = \emptyset$. Thus, we should prove that any node $w$ re-enters $Q$ only a finite number of times. Suppose $Q$ is about to be repartitioned. Let $w \not\in Q''$ be reachable from node $v \in Q''$. Since $G_d$ is acyclic, we have $d_v < d_w$. Thus, $d_w > \min\{d_u | u \in Q''\}$. Any node $u$ with the smallest label among all nodes in $Q''$ at the start of the repartitioning of $Q$, will never re-enter $Q$ (due to the positive arc costs). Therefore the number of repartitionings of $Q$ is bounded by $O(|V|)$ (like the Partitioning Shortest Path problem). Between two subsequent repartitionings, at most $O(|A|)$ arcs are scanned. Thus, the Topological Scan algorithm is still finite.

Furthermore, the addition of some extra nodes to $Q$ during the repartitioning of $Q$, does not influence the correctness of the algorithm. Suppose $Q$ is about to be repartitioned. Let $w \not\in Q''$ be reachable from node $v \in Q''$. By Definition 2.28, there is a path $(v = v_0, a_1, \ldots, v_{k-1}, a_k, v_k = w)$ in $G_d$. The topological sort on $Q'$ results in selecting node $v_{k-1}$ as a pivot node, before node $w$. After scanning arc $(v_{k-1}, w)$, label $d_w$ is surely finite. The correctness of the Generic Shortest Path algorithm does not change if some node with a finite label is re-selected as a pivot node.

**Proposition 3.25** The Topological Scan algorithm runs in $O(|V| |A|)$ time on a digraph $G = (V, A)$ with positive arc costs.

**Proof:** In the proof of Proposition 3.24, we showed that the number of repartitionings of $Q$ is bounded by $O(|V|)$ and that between two subsequent repartitionings at most $|A|$ arcs are examined. The repartitioning of $Q$ can be done in one Depth First Search, which takes $O(|V| + |A_d|)$ time. The Topological Scan algorithm therefore runs in $O(|V| (|V| + |A_d| + |A|)) = O(|V| |A|)$ time.

### 3.3.2.3 Small Label First

A simple strategy for placing nodes with small labels near the top of the queue is the Small Label First algorithm (or SLF for short), proposed by Bertsekas [18]. At each iteration, the node exiting $Q$ is the top node. Whenever a node $v$ enters $Q$, its label $d_v$ is compared with the label $d_w$ of the top node $w$ in $Q$. If $d_v \leq d_w$, node $v$ is entered at the top of $Q$. Otherwise, $v$ is entered at the bottom of $Q$.

**Proposition 3.26** [27] The number of iterations required by the Small Label First algorithm, is $\Theta(2^{|V|})$.

**Proof:** The number of iterations of the Generic Shortest Path algorithm is $O(2^{|V|})$ (see Theorem 3.4). To prove Proposition 3.26, we only have to find an example where this number of iterations actually occurs. Consider the graph family in Figure 3.5.
Suppose that, for all nodes, the list of outgoing arcs is sorted in descending order of arc costs. During the execution of Small Label First on a particular graph belonging to this family with node 0 as the origin, nodes $2k - 1$ and $2k$ ($1 \leq k \leq n$) are scanned exactly $2^{k-1}$ times. A detailed proof for this behavior can be found in [27].

![Figure 3.5: A family of pathological graphs for Small Label First. Some arc costs are not drawn in this picture. These can be found in Table 3.1.](image)

### Table 3.1: Arc costs for Figure 3.5, $1 \leq k < n$

<table>
<thead>
<tr>
<th>From node</th>
<th>To node</th>
<th>Arc cost</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>$10 \times 2^{n-1} - 3$</td>
</tr>
<tr>
<td>$2k - 1$</td>
<td>$2n + k$</td>
<td>$10 \times 2^{n-k} - 5$</td>
</tr>
<tr>
<td>$2k - 1$</td>
<td>$2k + 2$</td>
<td>$10 \times 2^{n-k} - 6$</td>
</tr>
<tr>
<td>$2k - 1$</td>
<td>$2k + 1$</td>
<td>$10 \times 2^{n-k-1} - 2$</td>
</tr>
<tr>
<td>$2k$</td>
<td>$2k + 2$</td>
<td>$10 \times 2^{n-k-1} - 3$</td>
</tr>
</tbody>
</table>

Like in the previously pathological graph for D’Esopo-Pape, this graph family is an unusual case, not only because of the arc lengths, but also due to the ordering of the adjacency lists. Chen and Powell [27] proved that Small Label First runs in polynomial time, if the list of outgoing arcs of every node is sorted in non-decreasing order of arc costs.

**Proposition 3.27** Let $G = (V, A)$ be a directed graph with nonnegative arc costs. If the list of outgoing arcs of any node $v \in V$ is sorted in non-decreasing order of the associated arc costs, Small Label First will run in $O(|V|^2 |A|)$ time.

**Proof:** Consider the queue $Q_1 = (v_1, v_2, \ldots, v_k)$ generated at the end of an iteration of Small Label First, on a graph where the lists of outgoing arcs for every node are ordered according to non-decreasing arc costs. In the next iteration, node $v_1$ is removed and scanned. Some nodes $\in V \setminus Q_1$ may be inserted in $Q_1$, either in front of node $v_2$ or behind node $v_k$, creating a new queue $(\ldots, v_2, v_3, \ldots, v_k, \ldots)$. At some
3.3 Label-correcting algorithms

later iteration, we will have a queue \( Q_i = (v_{i1}, v_{i+1}, \ldots, v_k, \ldots) \) \((i = 2, 3, \ldots, k)\), where \( Q_i \) is the first queue with top node \( v_i \) after scanning node \( v_1 \) of \( Q_1 \).

Starting with \( Q_1 \), node \( v_i \) is removed from \( Q_i \) and maybe some nodes \( w \) with \((v_i, w) \in \delta^+(v_i)\) are added to \( Q \) in front of \( v_{i+1} \), resulting in \( Q = (u_1, u_2, \ldots, u_h, v_{i+1}, v_{i+2}, \ldots, v_k, \ldots) \). By the definition of SMALL LABEL FIRST, we have \( d_{u1} \leq d_{u2} \leq \ldots \leq d_{uh} \).

Due to the ordering of the arcs in \( \delta^+(v_i) \), we also have \( d_{u1} \geq d_{u2} \geq \ldots \geq d_{uh} \).

Thus, \( d_{u1} = d_{u2} = \ldots = d_{uh} \). As long as the top node of \( Q \) is not equal to \( v_{i+1} \), another node \( w \) is chosen as pivot node and removed from \( Q \). If new nodes are added to the top of \( Q \), during the scanning of node \( w \), they must have a label equal to \( d_{uh} \) since the arc costs are nonnegative. Thus, starting from \( Q \), and before \( Q_{i+1} \)

is generated, the labels of the nodes inserted in front of node \( v_{i+1} \), are all the same. Hence, any node put in front of node \( v_{i+1} \) while transferring from \( Q_i \) to \( Q_{i+1} \), is scanned exactly once. Since only nodes that are not yet in \( Q_i \) can be added, at most \(|V\setminus Q_i|\) iterations are needed to go from \( Q_i \) to \( Q_{i+1} \).

Since going from \( Q_i \) to \( Q_{i+1} \) means that any node in \(|V\setminus Q_i|\) is scanned only once at most, each arc is examined at most once as well. So, at most \( O(|A|) \) basic operations are needed in order to go from \( Q_i \) to \( Q_{i+1} \).

Consider \( Q_1 \) and suppose that node \( v_{p}, 1 \leq p \leq k \), is a node with the smallest label among all nodes in \( Q_1 \). Clearly, node \( v_p \) will never enter any queue after it is scanned since the arc costs are nonnegative. Starting with \( Q_1 \), the algorithm will scan all nodes in \( Q_1 \) at least once before \( v_k \) is scanned. After scanning node \( v_k \), node \( v_p \) will not appear in any subsequent queue. As shown above, at most \( O(k|A|) \leq O(|V| |A|) \) operations are needed to go from \( Q_1 \) to \( Q_k \). At least one node \( v_p \) of \( Q_1 \) will never re-enter \( Q \). Since there are \(|V| \) nodes, at most \( O(|V|^2 |A|) \) operations are performed by SMALL LABEL FIRST.

In [20], a modification to SMALL LABEL FIRST is proposed to ensure that the running time is \( O(|V| |A|) \). This modification is based on the following idea: Suppose there is a set of non-decreasing iteration indices \( t_1, t_2, \ldots, |V| + 1 \) such that \( t_1 = 1 \), and for \( i = 1, \ldots, |V| \), all nodes that are in \( Q \) at the start of iteration \( t_i \) are removed from \( Q \) at least once prior to iteration \( t_{i+1} \). Because all arc lengths are nonnegative, this guarantees that any node having the smallest label among all nodes of \( Q \) at the start of iteration \( t_i \), will never re-enter \( Q \) after iteration \( t_{i+1} \). Thus, \( Q \) contains no more than \(|V| - i \) nodes at the start of iteration \( t_{i+1} \). Furthermore \( Q = \emptyset \) prior to iteration \( t_{|V|+1} \). If the running time between iterations \( t_i \) and \( t_{i+1} \) is \( O(|A|) \), the running time of the algorithm is \( O(|V| |A|) \). To bound the running time between iterations \( t_i \) and \( t_{i+1} \), SMALL LABEL FIRST is modified as follows. Given a positive \( k \in \mathbb{N}, k > 1 \), the iterations of the algorithm are separated in successive blocks of \( k|V| \) iterations each. Within each block of \( k|V| \) iterations, each node can be inserted at most \( k - 1 \) times at the top of \( Q \). Thus, if a node \( v \) is inserted \( k - 1 \) times to the top of \( Q \) within a certain block of \( k|V| \) iterations, all subsequent insertions of node \( v \) within that block of iterations are at the bottom of \( Q \) even if node \( v \) has a label not greater than the label of the top node of \( Q \). Now, between iterations \( t_i \) and \( t_{i+1} \), each node is scanned at most \( k - 1 \) times and each arc is thus examined \( k - 1 \)
times at most. Thus, the running time between iterations $t_i$ and $t_{i+1}$ is bounded by $O(|A|)$. In [20], it is shown that it is highly unlikely that the extra restriction will be exercised in practice if $k \geq 4$.

### 3.3.3 Some optimization strategies

#### 3.3.3.1 Sharp labels

While running the Generic Shortest Path algorithm (Algorithm 3.4), an outtree $\tilde{G} = (\tilde{V}, \tilde{A})$ rooted at node $s$ (see Definition 2.47) is maintained, with $\tilde{V} = \{ v | d_v \leq \infty \}$ and $\tilde{A} = \{ (\pi_v, v) | v \in \tilde{V} \setminus s \}$. In [179], the notion of sharp labels is introduced.

**Definition 3.28** While running the Generic Shortest Path algorithm, the label $d_v$ of node $v \in \tilde{V}$ is called sharp if, for any predecessor $u$ of node $v$ in the outtree $\tilde{G}$, the following equality holds: $d_u + \ell(P_{uv}) = d_v$ (where $P_{uv}$ denotes the (unique) $u$-$v$ path in the tree $\tilde{G}$).

We can see that choosing a node $v$ as pivot node in the Generic Shortest Path algorithm with a non-sharp label $d_v$, is undesirable. All label updates following directly or indirectly from node $v$ will be updated again.

Shier and Witzgall showed that, while running the algorithm of Gilsinn and Witzgall or D’Esopo-Pape, all selected pivot nodes have a sharp label. They also proved that this is not the case while running Bellman-Ford-Moore.

Glover and Klingman [86] refined the concept of sharp node labels:

**Definition 3.29** A specific implementation of the Generic Shortest Path algorithm is said to be:

- Globally sharp if all finite node labels are sharp, each time the (partial) Shortest Path Tree is updated.
- Globally scan-sharp if all finite node labels are sharp at the beginning of each iteration (as soon as a pivot node is selected).
- Locally scan-sharp if the selected pivot node has a sharp label just before the actual scanning starts.

Dial [58] introduced a method that ensures all pivot nodes to have a sharp label. As soon as an arc $(v, w)$ with $d_v + c_{vw} < d_w$ is scanned, $\delta = d_w - d_v - c_{vw}$ is determined. The label $d_u$ of any node $u$ in the subtree rooted at node $w$ is decreased by $\delta$. Furthermore, if $u \notin Q$, then node $u$ is added to $Q$. In [58], it is observed that the overhead required to ensure all pivot nodes will have a sharp label, outweighs the benefits of reducing the total number of iterations. The algorithm used is globally sharp, according to Definition 3.29.

In [86], the update of the labels in the subtree of node $w$ is postponed to the moment where all arcs in $\delta^+(v)$ are examined. The subtrees rooted at all nodes with an
improved label are then updated. The resulting algorithm is clearly a global scan-sharp algorithm. It might happen that more subtrees have to be traversed than in the globally sharp variant. But in the globally scan-sharp version, these subtrees are always mutually disjunct.

Desrochers [56] introduced a way to maintain local scan-sharpness. As soon as node \( v \) is selected as a pivot node, the \texttt{MakeLabelSharp} procedure (Algorithm 3.9) is executed with \( v \) as parameter. First, the difference \( \delta \) between \( d_v \) and the length of the \( s-v \) path in the current Shortest Path Tree is determined (lines 3–5). If \( \delta = 0 \), label \( d_v \) is sharp. If \( \delta > 0 \), label \( d_v \) is not sharp. It is possible that some of the predecessors of node \( v \) on the \( s-v \) path also have non-sharp labels. All these labels are updated and, if necessary, nodes are added to \( Q \) (line 13).

Desrochers originally added the \texttt{MakeLabelSharp} procedure to the Partitioning Shortest Path algorithm (see Section 3.3.1.5). In the proof of Proposition 3.20, it is stated that Partitioning Shortest Path needs at most \(|V|\) repartitionings. Between two repartitionings, at most \(|V|\) nodes are scanned and at most \(|A|\) arcs are examined. Including the \texttt{MakeLabelSharp} procedure, which runs in \( O(|V|) \), thus leads to a variant of the Partitioning Shortest Path algorithm with a worst case bound of \( O(|V|(|V|^2 + |A|)) = O(|V|^3) \). In Section 3.3.1.5, a better bound is established if the amount of nodes, transferred from \( Q'' \) to \( Q' \) during a repartitioning, is bounded by a certain constant \( n \) (independent of \(|V|\)) where at least one node with a minimum label among all nodes in \( Q'' \) is transferred to \( Q' \). The resulting bound is \( O(|V|^2) \).

\begin{algorithm}
\caption{\texttt{MakeLabelSharp}(\( v \))}
\begin{algorithmic}[1]
\State \( u \leftarrow v \)
\State \( \delta \leftarrow d_u \)
\While {\( u \neq s \)}
\State \( \delta \leftarrow \delta - c_{\pi_u,u} \)
\State \( u \leftarrow \pi_u \)
\EndWhile
\If {\( \delta > 0 \)}
\State \( d_v \leftarrow d_v - \delta \)
\State \( u \leftarrow v \)
\EndIf
\While {\( d_{\pi_u} + c_{\pi_u,u} > d_u \)}
\State \( d_{\pi_u} \leftarrow d_u - c_{\pi_u,u} \)
\State \( u \leftarrow \pi_u \)
\EndWhile
\If {\( u \notin Q \)}
\State \( Q \leftarrow Q \cup \{u\} \)
\EndIf
\end{algorithmic}
\end{algorithm}

### 3.3.3.2 Parent checking

Cherkassky [29] improved \texttt{Bellman-Ford-Moore} by introducing the \texttt{parent-check} strategy, which is in fact another way to skip nodes with a clearly non-sharp label rather than scanning them. The parent-check is executed immediately after having
different pivot node $v$. If $\pi_v \in Q$, then node $v$ is simply removed from $Q$ but the arcs in $\delta^+(v)$ are not scanned. Experimental evaluation showed that this heuristic is effective in combination with Bellman-Ford-Moore. Experiments on a large road map additionally showed that the parent-check indeed is effective if combined with Bellman-Ford-Moore and with Small Label First [121].

3.3.3.3 Large Label Last

In [20], a strategy is proposed that aims to choose a pivot node with a small label. At each iteration, when the node $v$ at the top of $Q$ has a label $d_v$ such that

$$d_v > \frac{\sum_{w \in Q} d_w}{|Q|},$$

node $v$ is not removed from $Q$ but is instead repositioned at the bottom of $Q$. Thus, nodes with a label larger than the average node label in $Q$ are moved to the last position of $Q$. This strategy is therefore called Large Label Last, or LLL for short.

It is easy to keep the sum of the labels of all nodes in $Q$ up to date during the execution of the Generic Shortest Path algorithm. The same goes for maintaining the cardinality of $Q$. Thus, the overhead to implement the Large Label Last strategy is minimal. This strategy can be combined with all label-correcting methods described. Experimental evaluations [121] showed that the running times of most label-correcting algorithms were reduced significantly when they are combined with the Large Label Last strategy.

3.3.4 Stop condition for label-correcting algorithms

When a node $v$ is removed from $Q$ for the first time in a label-correcting algorithm, its label $d_v$ does not necessarily equal $d(s,v)$, the length of a shortest $s$-$v$ path. The label $d_v$ might be improved and the node $v$ then re-enters $Q$. So for both the SSSD-SPP and the SSMD-SPP, we may not terminate the algorithm when only the nodes, to which the shortest path has to be computed, have been removed from $Q$ once, like we did in a label-setting algorithm (see Section 3.2.1).

However, also for label-correcting methods, a sharper stop condition is possible. Consider the SSMD-SPP with the set $\mathcal{T}$ of destination nodes. Since the labels $d_v$ are an upper bound for the length of a shortest path from $s$ to $v$, an upper bound for all desired shortest paths is $\max_{t \in \mathcal{T}} d_t$. We therefore only have to consider those nodes $v$ from $Q$, for which $d_v < \max_{t \in \mathcal{T}} d_t$, and proceed until $Q$ is empty, or, equivalently, only add nodes to $Q$ for which $d_v + c_{vw} < \min\{d_w, \max_{t \in \mathcal{T}} d_t\}$.
3.4 Arc set partition

Johnson [115] proposed an algorithm (see Algorithm 3.10) based on arc set partition. Let $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ be a directed graph and let $\mathcal{A}^1, \mathcal{A}^2, \ldots, \mathcal{A}^k$ be a partition of $\mathcal{A}$. This algorithm uses $k$ (non disjunct) candidate sets $Q^1, Q^2, \ldots, Q^k$. Each time the label $d_w$ of a node $w$ is updated, node $w$ is added to all $k$ candidate-sets.

Johnson’s algorithm can be seen as a generalization of Yen’s algorithm [203]. Yen used an arbitrary ordering of the node set to partition the graph into two directed acyclic graphs. Let $\mathcal{V} = \{v_1, v_2, \ldots, v_{|\mathcal{V}|}\}$ be the node set. The arc set $\mathcal{A}$ is partitioned into $\mathcal{A}^1 = \{(v_i, v_j) | (v_i, v_j) \in \mathcal{A}, j > i\}$ and $\mathcal{A}^2 = \{(v_i, v_j) | (v_i, v_j) \in \mathcal{A}, j < i\}$. Clearly, $\mathcal{G}^1 = (\mathcal{V}, \mathcal{A}^1)$ and $\mathcal{G}^2 = (\mathcal{V}, \mathcal{A}^2)$ are directed acyclic graphs. Originally, Yen used those two arc sets to improve Bellman-Ford-Moore (see Section 3.3.1.1) and proved that the overall number of operations is reduced by half. Recently, Banister and Eppstein [7] proved that, if a randomized ordering of the node set is used, an additional expected speedup of $2/3$ is achieved.

Algorithm 3.10: Johnson’s Generic Shortest Path ($\mathcal{G}, s$)

1: INITIALIZE-SINGLE-SOURCE($\mathcal{G}, s, Q^1$)
2: for $i \in [2, 3, \ldots, k]$ do
3: \hspace{1cm} $Q^i = \{s\}$
4: repeat
5: \hspace{1cm} for $i \in [1, 2, \ldots, k]$ do
6: \hspace{2cm} if $Q^i \neq \emptyset$ then
7: \hspace{3cm} Select a pivot node $v$ from $Q^i$
8: \hspace{3cm} $Q^i \leftarrow Q^i - \{v\}$
9: \hspace{2cm} foreach $(v, w) \in A^i$ do
10: \hspace{3cm} \hspace{1cm} if $d_v + c_{vw} < d_w$ then
11: \hspace{4cm} $d_w \leftarrow d_v + c_{vw}$
12: \hspace{4cm} $\pi_w \leftarrow v$
13: \hspace{3cm} \hspace{1cm} for $j \in [1, 2, \ldots, k]$ do
14: \hspace{4cm} \hspace{1cm} if $w \notin Q^j$ then $Q^j \leftarrow Q^j \cup \{w\}$
15: \hspace{2cm} until $Q^i = \emptyset$
16: until $\bigcup_{i=1}^k Q^i = \emptyset$
All the shortest path problems mentioned in Chapter 3 search unidirectionally from origin $s$. Especially if we consider the SSSD-SPP, it is more efficient in practice to search bidirectionally, using both the origin $s$ and the destination $t$ uniformly by searching alternatively or simultaneously from the origin side, as well as from the destination side [41, 61, 140, 157].

Because the searched area now not consists of one big ‘circle’ with a radius equal to the distance between $s$ and $t$, but of two smaller ‘circles’ with each a radius of about half the distance between $s$ and $t$, the number of nodes scanned will be reduced to approximately half the original number (see Figure 4.1). Moreover if one can use a parallel implementation, the running time will be roughly a quarter of the running time of a unidirectional implementation.

\begin{figure}
\centering
\includegraphics[width=\textwidth]{fig41.png}
\caption{Unidirectional search (left) and bidirectional search (right)}
\end{figure}
4.1 Notation

In this chapter, we are interested in the shortest path from node \( s \) to node \( t \) in the graph \( G = (V, A) \). The forward search takes node \( s \) as origin. The forward search is based on the graph \( G^s = (V^s, A^s) \), which is equal to the original graph \( G = (V, A) \). The superscript \( s \) is added to distinguish this graph from the one used in the backward search. The backward search takes node \( t \) as origin and is based on the graph \( G^t = (V^t, A^t) \). The graph \( G^t \) is actually \( G^s \) reversed, thus \((v, w) \in A^t\) if and only if \((w, v) \in A^s\). Furthermore, this arc has a related arc cost \( c^t_{vw} = c^s_{wv} \).

Although the node sets of the graphs \( G^s \) and \( G^t \) are equal, we use the notation \( V^s \) and \( V^t \). A node \( v \in V^s \) has an associated label \( d^s_v \) and a predecessor \( \pi^s_v \). In the backward search for node \( v \in V^t \), the label \( d^t_v \) and the predecessor \( \pi^t_v \) are used.

Regarding bidirectional search, we refer to the two processes (forward search and backward search) as the primary and the opposite process. Either side, whether it uses \( s \) or \( t \) as origin, may be appointed as primary. In the opposite process, everything is denoted by a tilde. So \( \tilde{d}^s_v = d^t_v \), \( \tilde{d}^t_v = d^s_v \), \( \tilde{Q}^s = Q^t \), \( \tilde{Q}^t = Q^s \), etc.

4.2 Bidirectional label-setting

For bidirectional label-setting methods, we introduce some additional notation: \( R^s = \{ v \in V^s | d^s_v \neq \infty \land v \notin Q^s \} \) contains the nodes settled in the forward search. For a node \( v \in R^s \), the shortest path length from node \( s \) to node \( v \) equals \( d^s_v \). Similarly, \( R^t = \{ v \in V^t | d^t_v \neq \infty \land v \notin Q^t \} \) contains all the nodes settled in the backward search.

In each iteration of this algorithm, either a forward or a backward search iteration is performed. In Algorithm 4.1 such an iteration is described in terms of the primary process. Recall that entities of the opposite process are denoted by a tilde. Since we consider a label-setting algorithm, we choose a node with minimum label in the candidate set \( Q \) as pivot node (line 1). After removing node \( v \) from the candidate set, we check whether the opposite process has already scanned this node or not (line 3). If node \( v \) is scanned in both processes, both search spaces meet each other in node \( v \). We store this meeting node as \( u \) if \( u = \text{NIL} \) or \( d^s_u + d^t_v < d^s_v + d^t_u \).

Algorithm 4.2 shows a sequential implementation of a bidirectional label-setting algorithm. A selection rule has to be implemented in line 7 and a stop condition has to be provided in line 11.

4.2.1 Stop condition

**Proposition 4.1** During the execution of Algorithm 4.2 on a strongly connected graph \( G \), at least one meeting node \( u \) will be found.
Algorithm 4.1: \textsc{Iterate}(G, Q, u) (label-setting)

1: Let $v \leftarrow \arg\min \{d_v | v \in Q\}$
2: $Q \leftarrow Q - \{v\}$
3: if $v \in \tilde{R}$ then
4: if $u = \text{NIL}$ or $d_u^s + d_v^t < d_u^t + d_v^s$ then
5: $u \leftarrow v$
6: else
7: foreach $(v, w) \in \delta^+(v)$ do
8: \textsc{Label-Update}(v, w, Q)

Algorithm 4.2: \textsc{Bidirectional Shortest Path}(G*, Gt, s, t)

1: $u \leftarrow \text{NIL}$
2: \textsc{Initialize-Single-Source}(G*, s, Q*)
3: \textsc{Initialize-Single-Source}(Gt, t, Qt)
4: \textsc{Iterate}(G*, Q*)
5: \textsc{Iterate}(Gt, Qt)
6: repeat
7: if a certain selection rule holds then
8: \textsc{Iterate}(G*, Q*)
9: else
10: \textsc{Iterate}(Gt, Qt)
11: until a certain stop condition holds
**Proof**: Since we assume that graph $G$ is strongly connected, we know there is at least one $s$-$t$ path. In Algorithm 4.2, an initial forward iteration is executed in line 4 and an initial backward iteration is executed in line 5 to make clear that, even if the decision rule in line 7 only considers one particular search direction, we will find a meeting node $u$ eventually. Recall that a meeting node $u$ has to be settled in both directions, thus $u \in \mathcal{R}^s \cap \mathcal{R}^t$. Executing one initial iteration in both directions ensures that $\mathcal{R}^s \neq \emptyset$ and $\mathcal{R}^t \neq \emptyset$, so eventually we will find at least one meeting node $u$.

The stop condition in a bidirectional algorithm turned out to be difficult. Nicholson [140] used the following stop condition, which he proved to be correct:

$$
\min_{v \in (\mathcal{R}^s \cup \mathcal{Q}^s) \cap (\mathcal{R}^t \cup \mathcal{Q}^t)} d_v^s + d_v^t \leq \min_{v \in \mathcal{Q}^s} d_v^s + \min_{v \in \mathcal{Q}^t} d_v^t.
$$

(4.1)

Dreyfus [61] improved stop condition (4.1) as follows: Stop as soon as a meeting node $u$ is found the first time. He mentioned earlier work by others on bidirectional algorithms, in which it is erroneously assumed that the shortest path always has a length $d_u^s + d_u^t$. He proved that the shortest path has length:

$$
\min \{ d_u^s + d_u^t, \min \{ d_v^s + c_{vw} + d_w^t | v \in \mathcal{R}^s, w \in \mathcal{R}^t, (v,w) \in A \} \}.
$$

(4.2)

**Theorem 4.2** If $u \in \mathcal{R}^s \cap \mathcal{R}^t$, the shortest path length from $s$ to $t$ is given by

$$
\min \{ d_u^s + d_u^t, \min \{ d_v^s + c_{vw} + d_w^t | v \in \mathcal{R}^s, w \in \mathcal{R}^t, (v,w) \in A \} \}.
$$

**Proof**: Let $u \in \mathcal{R}^s \cap \mathcal{R}^t$, $(v,w) \in A$, $q \notin \mathcal{R}^s \cup \mathcal{R}^t$ be any node such that $(v,q) \in A$ and $r \notin \mathcal{R}^s \cup \mathcal{R}^t$ be any node such that $(r,w) \in A$ (see Figure 4.2).

![Explanatory diagram with Theorem 4.2](image)

---

1In [157], it was abusively stated that the left-hand side of inequality (4.1) is $\min \{ d_v^s + d_v^t | v \in \mathcal{R}^s \cap \mathcal{R}^t \}$. 

4.2 Bidirectional label-setting

We have \( d_s^u \leq d_s^v + c_{vq} \), since \( u \in \mathcal{R}^s \) and \( q \notin \mathcal{R}^s \). Also, \( d_t^u \leq d_t^w + c_{rw} \), since \( u \in \mathcal{R}^t \) and \( r \notin \mathcal{R}^t \). This gives \( d_s^u + d_t^u \leq d_s^v + c_{vq} + c_{rw} + d_t^w \). So any path containing a node \( q \notin \mathcal{R}^s \cup \mathcal{R}^t \) is at least as long as the path via meeting node \( u \).

However, it is possible that \( d_s^u > d_s^v \) and \( d_t^u > d_t^w \). Hence, it is possible that \( d_s^u + d_t^u > d_s^v + c_{vq} + c_{rw} + d_t^w \). Consider for example the case \( d_s^u = d_t^u = 10, d_s^v = d_t^w = 8 \) and \( c_{vw} = 3 \).

Algorithm 4.3: Post-Phase(\( G^s, G^t, u \)) (label-setting)

1: if \(|Q^s| \leq |Q^t|\) then
2:   foreach \( v \in Q^s \) do
3:     if \( d_s^v + d_t^v < d_s^u + d_t^u \) then \( u \leftarrow v \)
4:   else
5:     foreach \( v \in Q^t \) do
6:       if \( d_s^v + d_t^v < d_s^u + d_t^u \) then \( u \leftarrow v \)

Ensure: \( d_s^u + d_t^u \) is the length of the shortest \( s-t \) path

As a result of Theorem 4.2, it is obvious that stop condition (4.2) is valid. The set \( Q^t = \{ v | (v, w) \in A, v \notin \mathcal{R}^t, w \in \mathcal{R}^t \} \), thus \( \min \{ d_v^s + c_{vw} + d_w^t \} \) can be reformulated as \( \min \{ d_v^s + c_{vw} + d_w^t | v \in \mathcal{R}^s, w \in \mathcal{R}^t, (v, w) \in A \} \). Furthermore, we can see that (4.2) can also be rewritten as \( \min \{ d_v^s + d_w^t | v \in \mathcal{R}^s \cap Q^t \} \). To implement Dreyfus’ stop condition, we stop as soon as a meeting node \( u \in \mathcal{R}^s \cap \mathcal{R}^t \) is found. At the end of the algorithm, we need to run a post-phase (see Algorithm 4.3). To minimize the running time, we decide in line 1 to iterate over the nodes in either \( Q^s \) or \( Q^t \), depending on the cardinality of these sets.

In [96], a new stop condition is presented, which incorporates the Post-Phase routine directly into Algorithm 4.2. This stop condition is:

\[
\min_{v \in Q^s} d_v^s + \min_{v \in Q^t} d_v^t > d_s^u + d_t^u. \tag{4.3}
\]

In order to use this stop condition, we need to update \( u \) in a different manner (observe Algorithm 4.4). Now, \( u \) is updated as soon as an \( s-t \) path via arc \((v, w)\) is found that is shorter than all other \( s-t \) paths found so far.

It is easy to see that \( u \in (\mathcal{R}^s \cup Q^s) \cap (\mathcal{R}^t \cup Q^t) \) and that the stop condition (4.3) equals the stop condition of Nicholson (4.1). The way stop condition (4.3) is presented in [96] leads to an efficient implementation (see Algorithm 4.4). Instead of keeping track of a shortest path found so far, via meeting nodes \( u \in \mathcal{R}^s \cap \mathcal{R}^t \) only, all paths via meeting nodes \( u \in (\mathcal{R}^s \cup Q^s) \cap (\mathcal{R}^t \cup Q^t) \) are now taken into account to determine the shortest path found so far.
Algorithm 4.4: Iterate (version 2)($G, Q$) (label-setting)

1: Let $v \leftarrow \arg \min \{d_v | v \in Q\}$
2: $Q \leftarrow Q - \{v\}$
3: foreach $(v, w) \in \delta^+(v)$ do
4: \hspace{1em} Label-Update($v, w, Q$)
5: if $w \in \tilde{R}$ then
6: \hspace{2em} if $u = \text{NIL}$ or $d_w + \tilde{d}_w < d_u + \tilde{d}_u$ then
7: \hspace{3em} $u \leftarrow w$

4.2.2 Forward or backward?

In line 7 of Algorithm 4.2, a certain decision rule is mentioned to decide whether a forward or backward iteration has to be performed. Any rule results in a correct algorithm, as we will see later. So we try to find the most efficient decision rule. If the decision rule is computationally complicated, it will increase the running time of the algorithm since the decision rule is executed with each iteration. We mention some commonly used rules:

Rule 1: Choose forward search only. This results in a unidirectional label-setting algorithm, as described in Section 3.2.

Rule 2: Alternate between forward search and backward search (Dantzig [41]).

Rule 3: Execute a forward search iteration if $\min \{d_v | v \in Q^s\} \leq \min \{d_v | v \in Q^f\}$ and a backward iteration otherwise (Nicholson [140]).

Rule 4: Execute a forward search iteration if $|Q^s| \leq |Q^f|$ and a backward iteration otherwise (Pohl [157]).

Rule 1 is mentioned to show the relation between unidirectional and bidirectional search methods. Rules 2 and 3 implicitly hypothesize a uniform distribution of nodes and arcs in the graph. Rule 4 is called the cardinality comparison rule. The sizes $|Q^s|$ and $|Q^f|$ reflect the density of the respective forward and backward search spaces. Expanding the direction with the smallest number of candidates means making progress in a sparser region.

Table 4.1: Cumulated reached nodes for a set of 100 SPP instances

<table>
<thead>
<tr>
<th>Rule</th>
<th>Forward</th>
<th>Backward</th>
<th>Total</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (Unidirectional)</td>
<td>8,346,242</td>
<td>451</td>
<td>8,346,693</td>
<td>100.0</td>
</tr>
<tr>
<td>2 Dantzig [41]</td>
<td>2,706,935</td>
<td>2,707,121</td>
<td>5,414,056</td>
<td>64.9</td>
</tr>
<tr>
<td>3 Nicholson [140]</td>
<td>2,850,144</td>
<td>2,875,502</td>
<td>5,725,646</td>
<td>68.6</td>
</tr>
<tr>
<td>4 Pohl [157]</td>
<td>2,737,753</td>
<td>2,439,406</td>
<td>5,177,159</td>
<td>62.0</td>
</tr>
</tbody>
</table>

In Table 4.1, we show for each rule the cumulated nodes reached for a set of 100 SSSD-SPP instances on a road map. Even for Rule 1 (unidirectional forward search), we have some nodes reached in the backward direction due to the initial backward
4.3 Bidirectional label-correcting

The bidirectional algorithm described in Algorithm 4.2, can be applied to any of the label-correcting algorithms as well [121]. The idea is that, once a node \( u \) is labeled both from \( s \) and from \( t \), the sum of these label values (\( d_u^s + d_u^t \)) is an upper bound for the shortest path from \( s \) to \( t \). We take \( d_{\text{max}}^u = d_u^s \) and \( d_{\text{max}}^u = d_u^t \) as maximum values for nodes labeled from \( s \) and \( t \), respectively. If, during a forward iteration, \( v \in Q^s \) is selected as pivot node and \( d_v^s \geq d_{\text{max}}^s \), we simply ignore \( v \) instead of scanning its outgoing arcs. The same holds for a backward iteration: if node \( v \in Q^t \) is selected as pivot node and \( d_v^t \geq d_{\text{max}}^t \), its incoming arcs are not scanned. During the initialization of Algorithm 4.2, we set the bounds \( d_{\text{max}}^s \leftarrow \infty \) and \( d_{\text{max}}^t \leftarrow \infty \).

Algorithm 4.5 shows the iteration procedure for a label-correcting algorithm. In line 1, a pivot node \( v \in Q \) is selected. After removing the pivot node from \( Q \), it is checked whether the pivot node \( v \) is labeled from both sides or not (line 3). If the pivot node \( v \) is labeled from both sides, it is checked whether or not the bounds \( d_{\text{max}}^s \) and \( d_{\text{max}}^t \) can be improved. If so, node \( v \) is stored as meeting node \( u \). Note that in the primary process, we do not consider an improvement of the primary bound \( d_{\text{max}} \) if \( \tilde{d}_v > d_{\text{max}} \). In line 8, it is checked whether or not we can omit scanning the outgoing arcs of \( v \).

\text{Algorithm 4.5: } \text{iterate}(G, Q) \ (\text{ labeling-correcting })

1: Choose a pivot node \( v \in Q \)
2: \( Q \leftarrow Q - \{v\} \)
3: if \( \tilde{d}_v < \infty \) then
4: \hspace{1em} if \( d_v^s \leq d_{\text{max}}^s \) and \( d_v^t \leq d_{\text{max}}^t \) then
5: \hspace{2em} \( d_{\text{max}}^s \leftarrow d_v^s \)
6: \hspace{2em} \( d_{\text{max}}^t \leftarrow d_v^t \)
7: \hspace{1em} \( u \leftarrow v \)
8: if \( d_v < d_{\text{max}} \) then
9: \hspace{1em} foreach \( (v, w) \in \delta^+(v) \) do
10: \hspace{2em} \text{label-update}(v, w, Q)

Recall that the proof of Theorem 4.2 is based on the property that, after termination of the bidirectional label-setting algorithm with a meeting node \( u \) having permanent labels \( d_u^s \) and \( d_u^t \), all labels \( d_v^s \leq d_u^s \) and all labels \( d_v^t \leq d_u^t \) are permanent. Furthermore, all labels \( d_v^s > d_u^s \) will never become smaller than \( d_u^s \). The same holds for the opposite direction: no labels \( d_v^t > d_u^t \) will become smaller than \( d_u^t \). This property also holds for a bidirectional label-correcting algorithm as described above. The algorithm stops when there are no more nodes left in queues \( Q^s \) and
Q'. At that moment, all labels $d_s^v \leq d_s^\text{max}$ and $d_t^v \leq d_t^\text{max}$ are permanent. Define $R^s = \{ v | d_s^v \leq d_s^\text{max} \}$ and $R^t = \{ v | d_t^v \leq d_t^\text{max} \}$. We have $u \in R^s \cap R^t$. Thus, Theorem 4.2 is also valid for the bidirectional label-correcting algorithm.

The post-phase routine for a bidirectional label-correcting routine (Algorithm 4.6) is computationally a little more expensive than for label-setting (Algorithm 4.3), because we cannot use the sets $Q^s$ and $Q^t$ since both are empty.

**Algorithm 4.6: Post-Phase($G^s, G^t, u$) (label-correcting)**

1: foreach $v \in V$ do
2: if $d_s^v + d_t^v < d_s^u + d_t^u$ then
3: $u \leftarrow v$

**Ensure:** $d_s^u + d_t^u$ is the length of the shortest s-t path

In Section 4.2.2, several alternation rules are described to determine if a forward or backward iteration has to be performed. Most of these rules can be applied to bidirectional label-correcting algorithms too. The rule of Nicholson [140] is based on the values $\min \{ d_s^v | v \in Q^s \}$ and $\min \{ d_t^v | v \in Q^t \}$. Since label-correcting algorithms are based on the idea that it is computationally too expensive to determine the node $v \in Q$ with $d_v = \min \{ d_v | v \in Q \}$, it makes no sense to implement the alternation rule of Nicholson in a bidirectional label-correcting algorithm. In a label-correcting algorithm, the pivot node $v$ is chosen to be the first node of the candidate queue $Q$. In a bidirectional label-correcting environment, we interpret Nicholson’s alternation rule similarly. Let $v$ be the first node of $Q^s$ and let $w$ be the first node of $Q^t$. Perform a forward iteration if $d_s^v \leq d_t^w$, and a backward iteration otherwise.

As far as we know, combining label-correcting algorithms and bidirectional search is first described by us in [121]. As we will see in the next chapter, bidirectional search is often combined with so-called guided search, where nodes that are likely to lead towards the destination node are scanned first. For guided search, it is even more complex to find a stop-condition. We will provide several (new) stop conditions in that chapter.
5

Heuristic estimators

Hart, Nilsson and Raphael [103] added a heuristic approach to the SPP. A heuristic approach typically uses specific knowledge about the problem. In this chapter, we describe their algorithm A* and its extensions to label-correcting and bidirectional variants.

5.1 The family of A* algorithms

Suppose we want to determine the shortest path from node \( s \) to node \( t \). During the execution of the Generic Shortest Path problem, a pivot node \( v \) is selected in each iteration. If it is obvious that \( v \) cannot be on a shortest path, it is a waste of effort to scan the outgoing arcs from node \( v \). To determine whether or not a node \( v \) might be on a shortest path from node \( s \) to \( t \), some additional information is used in the form of a heuristic estimator.

**Definition 5.1** A heuristic estimator \( h : \mathcal{V} \to \mathbb{R} \) is a function, providing \( h(v) \) as an estimate of the shortest path distance \( d(v, t) \) from node \( v \) to node \( t \).

Formally, \( h(v) \) is only an estimate of the shortest path distance \( d(v, t) \). So both \( h(v) \leq d(v, t) \) and \( h(v) > d(v, t) \) are possible.

In [103], such a heuristic estimator is incorporated in the so-called A* algorithm, as presented in Algorithm 5.1.

Algorithm 5.1 in fact describes a family of algorithms. The choice of a particular heuristic estimator \( h \) selects a particular algorithm from this family. So, if we refer to the A* algorithm, we mean the particular algorithm that is instantiated by a certain heuristic estimator \( h \).

**Theorem 5.2** [103] At the beginning of any iteration of the A* algorithm (on a graph with nonnegative arc lengths), the following condition holds for the set \( \mathcal{R} = \)
Algorithm 5.1: $A^*$ ($G, s, t$)

1: INITIALIZE-SINGLE-SOURCE($G, s, Q$)
2: repeat
3: Select a pivot node $v = \arg \min \{d_v + h_v | v \in Q\}$
4: if $v \neq t$ then
5: \hspace{1em} $Q \leftarrow Q - \{v\}$
6: \hspace{1em} foreach $(v, w) \in \delta^+(v)$ do
7: \hspace{2em} LABEL-UPDATE($v, w, Q$)
8: until $v = t$

{$v | d_v < \infty, v \notin Q$: For any node $v \notin R$ and any shortest $s$-$v$ path $P$, there exists a node $w \in P$ with $w \in Q$ and $d_w = d(s, w)$.

Proof: Let $P = \{v_0, (v_1, v_1), v_1, \ldots, (v_k-1, v_k), v_k\}$ with $s = v_0$ and $v_k = v$. At the beginning of the first iteration, we have $s \in Q$ and $d_s = d(s, s) = 0$. Thus, the condition trivially holds for $w = s$. At the beginning of all other iterations, we have $s \in R$, since $s$ enters $R$ during the first iteration and never leaves $R$ due to the nonnegative arc lengths. Let $V'$ be the set of all nodes $v_i \in P$ with $v_i \in R$ and $d_{v_i} = d(s, v_i)$. $V'$ is not empty, since $s \in V'$. Let $u$ be the highest index node in $V'$. Clearly, $u \neq v$ as $v \notin R$. Let $w$ be the successor of $u$ on $P$. Now, we have:

$$d_w \leq d_u + c_{uw} = d(s, u) + c_{uw} = d(s, w). \tag{5.1}$$

The inequality in (5.1) holds, since $u \in R$ and thus LABEL-UPDATE($u, w, Q$) is executed. The first equality holds by the definition of $V'$: since $u \in V'$, we have $d_u = d(s, u)$. The last equality holds since $P$ is a shortest $s$-$v$ path. By Theorem 3.1 part (i), label $d_w$ equals the length of an $s$-$w$ path and so $d_w \geq d(s, w)$. This implies that $d_w = d(s, w) < \infty$. Since $u$ is the node with the highest index in $V'$, we have $w \notin V'$. Hence, $w \in Q$.}

From here on we may sometimes need heuristic estimators where the estimates always form a lower bound on the actual distances.

Definition 5.3 A heuristic estimator $h$, such that $h_v \leq d(v, t)$ for all $v \in V$, is called optimistic or admissible.

In related literature, both terms optimistic and admissible can be found regularly. We prefer using the term optimistic, since it is more descriptive.

Theorem 5.4 [103] Let $G$ be a strongly connected graph with nonnegative arc lengths. Let $h$ be an optimistic heuristic estimator. Then, the $A^*$ algorithm terminates and, upon termination, $d_t$ is the length of a shortest $s$-$t$ path.

Proof: Algorithm 5.1 is a specification of the General Shortest Path algorithm, with a sharper stop condition. As soon as $v = t$ is selected as pivot node, we might
have $Q \neq \emptyset$. Applying Theorem 3.2, we conclude that the number of iterations in Algorithm 5.1 is finite. But we still have to prove that, upon termination, $d_t$ equals the length of a shortest $s$-$t$ path. As long as $A^*$ has not terminated, Theorem 5.2 states that for any shortest $s$-$t$ path $P$, there exists a node $v \in P$ with $v \in Q$ and $d_v = d(s, v)$. For node $v$, we have:

$$d_v + h_v = d(s, v) + h_v \leq d(s, v) + d(v, t) = d(s, t). \quad (5.2)$$

The inequality in (5.2), holds since $h_v \leq d(v, t)$ for all $v \in V$. The last equality in (5.2) holds because $v \in P$ and $P$ is a shortest $s$-$t$ path. Now, suppose that, upon termination, $d_t > d(s, t)$. In the final iteration, $t$ is selected as pivot node.

At that moment, a shortest $s$-$t$ path contains a node $v \in Q$ with the property: $d_v + h_v \leq d(s, t) < d_t = d_v + h_v$. Thus, at this stage, node $v$ would have been selected as pivot node rather than $t$, contradicting the assumption that $A^*$ has terminated.

In Definition 5.1, $h_v$ is defined as an estimate of $d(v, t)$ (not necessarily optimistic). If for some node $v \in V$ the heuristic estimate $h_v > d(v, t)$, then $d_t$ will not necessarily be equal to $d(s, t)$ upon termination of the $A^*$ algorithm. Therefore, we will commonly restrict ourselves to optimistic heuristic estimators. The $A^*$ algorithm is also used in areas where an approximate shortest path is acceptable. In such an environment, the use of a heuristic estimator where $h_v > d(v, t)$ for several nodes $v$ might improve the running time needed to find an approximate shortest path.

### 5.1.1 Consistent heuristic estimators

**Definition 5.5** A heuristic estimator $h$ is called consistent if and only if $h_v - h_w \leq d(v, w)$ for all pairs of nodes $v, w \in V$.

In related literature, sometimes a different definition is found: $h_v - h_w \leq c_{vw}$ for all $(v, w) \in A$.

**Proposition 5.6** The condition $h_v - h_w \leq d(v, w)$ for all pairs of nodes $v, w \in A$ in Definition 5.5 is equivalent to the condition $h_v - h_w \leq c_{vw}$ for all $(v, w) \in A$.

**Proof:** Suppose $h_v - h_w \leq d(v, w)$ for all pairs of nodes $v, w \in V$. Let $(v, w) \in A$ be given. Since $d(v, w) \leq c_{vw}$, we have $h_v - h_w \leq c_{vw}$. To prove the opposite, suppose $h_v - h_w \leq c_{vw}$ for all $(v, w) \in A$. Let a pair of nodes $v, w \in V$ be given. By definition $d(v, w)$ equals the length of some shortest path $P = (v = u_0, (u_0, u_1), u_1, \ldots, (u_{k-1}, u_k), u_k = w)$ from $v$ to $w$. For all arcs $(u_i, u_{i+1}) \in P$ ($0 \leq i \leq k - 1$), we have $h_{u_i} - h_{u_{i+1}} \leq c_{u_i, u_{i+1}}$. Adding these inequalities results in $h_{u_0} - h_{u_k} = h_v - h_w \leq \sum_{i=0}^{k-1} c_{u_i, u_{i+1}} = d(v, w)$. This completes the proof.

**Theorem 5.7** Let $G$ be a graph with nonnegative arc lengths. Let $h$ be a consistent heuristic estimator. Then, for the $A^*$ algorithm, the following condition holds for the set $R = \{v | d_v < \infty, v \notin Q\}$: a node $v \in R$ never re-enters the candidate set $Q$.
Proof: We prove this by contradiction. Suppose that, at some iteration, pivot node $v$ is selected and during the execution of the Label-Update subroutine, the label $d_w$ of a node $w \in \mathcal{R}$ is going to be decreased (and thus $w$ will be added again to $\mathcal{Q}$). Thus $d_w > d_v + c_{vw}$, or:
\[ d_w - d_v > c_{vw}. \] (5.3)

Since $w$ is selected as a pivot node in an earlier iteration, we have $d_w + h_w \leq d_v + h_v$. Because $h$ is a consistent heuristic estimator, we have:
\[ d_w - d_v \leq h_v - h_w \leq d(v, w) \leq c_{vw}. \] (5.4)

Clearly, (5.3) contradicts (5.4). It is therefore impossible that the label of a node $w \in \mathcal{R}$ can be decreased.

As a corollary of Theorem 5.7, the $A^*$ algorithm that uses a consistent heuristic estimator, belongs to the family of label-setting algorithms as is described in Section 3.2. Furthermore, if we use a consistent heuristic estimator, the $A^*$ algorithm will determine a shortest path from $s$ to $t$.

It is easy to see that if a heuristic estimator $h$ is consistent, so is $h + c$ for any constant $c$. Therefore, a consistent heuristic estimator $h$ might not always satisfy the condition $h_v \leq d(v, t)$. Thus, the condition $h_v \leq d(v, t)$ is sufficient, but not necessary for finding a shortest path with the $A^*$ algorithm.

Suppose a known set $\mathcal{H}$ of consistent heuristic estimators. The following theorem [169] shows a way to get the highest estimates from $\mathcal{H}$.

**Theorem 5.8** Let $\mathcal{H}$ be a set of consistent heuristic estimators. Then, the estimator $h^*$ defined by
\[ h^*_v = \max_{h \in \mathcal{H}} h_v \]

is consistent.

Proof: [201] It is obvious that $h^*_v \leq d(v, t)$. We have to prove that $h^*_v - h^*_w \leq d(v, w)$ for all pairs of nodes $v, w \in \mathcal{V}$. Given two nodes $v, w \in \mathcal{V}$, we have $h_v - h_w \leq d(v, w)$ for all $h \in \mathcal{H}$. Let $\tilde{h}_v = \arg \max_{h \in \mathcal{H}} h_v$:
\[ h^*_v - h^*_w = \max_{h \in \mathcal{H}} h_v - \max_{h \in \mathcal{H}} h_w = \tilde{h}_v - \max_{h \in \mathcal{H}} h_w \leq \tilde{h}_v - \tilde{h}_w \leq d(v, w). \]

Thus, $h^*$ is a consistent heuristic estimator. $\square$

5.1.2 The star in $A^*$

One may wonder where the name $A^*$ comes from. Algorithm 5.1 was first called $A^*$ by Hart, Nilsson and Raphael [103]. In 1980 however, Nilsson [142] changed
the name of Algorithm 5.1 to Algorithm A and he declared the A* algorithm to be a special case of Algorithm A, with the additional property that an optimistic heuristic estimator is used. Many researchers (e.g. Dechter and Pearl [46]) prefer to identify an algorithm by how it processes its input, rather than by the type of input it may encounter. In most related literature, Algorithm 5.1 is thus still called A*, regardless of the heuristic estimator that is actually used.

We do have an idea why Nilsson wanted to restrict the possible heuristic estimators for the A* family of algorithms to the set of all optimistic heuristic estimators. Much research is done on the so-called optimality of the A* algorithm. Here, optimality definitely has another meaning than we are used to. We prefer to use the term optimal to denote that an algorithm actually solves a problem to optimality. In the literature discussed in this section, the term optimality is used to denote a kind of superiority over other algorithms. If A* is said to be optimal, it is meant that A* scans no more nodes than any other equally informed algorithm (see Definition 5.9).

Although we are not particularly interested in whether or not A* is optimal in this sense, we provide a short overview of the research that is done on this so-called optimality of A*, since that is what the name A* is based on.

First, we need to understand the concept of equally informed algorithms:

**Definition 5.9** Two algorithms are called equally informed if they possess precisely the same knowledge of a problem as input, and if no other external information is added during the execution of the algorithms.

In the original paper [103], a first attempt is made to ‘prove’ that among all equally informed algorithms, the number of nodes scanned by A* using a consistent heuristic estimator, is minimal. The main argument was that, if some algorithm B (upon termination) did not scan a node v that is scanned by A*, then algorithm B must have known that any path from the origin node s to the destination node t via node v is not optimal. Since node v is scanned by A*, we have $d_v + h_v < d_t$. Thus, A* was not aware that any s-t path via node v is not optimal. Therefore, Algorithm B must have obtained additional information from some external source, unavailable to A*. Hence, algorithm B is more informed than A* and the comparison is unfair.

Of course, it might happen that in line 3 of Algorithm 5.1, several nodes in the candidate set $Q$ minimize $d_v + h_v$. In that case, a tie-breaking rule should be used to select one particular pivot node. The number of nodes scanned by the A* algorithm depends on this tie-breaking rule. Instead of claiming that A* is optimal among all equally informed algorithms, it was stated in [103] that for any problem instance, there exists a tie-breaking rule, which results in a smaller or equal number of scanned nodes when assigned to the A* algorithm.

Later [104], it was observed that the former argument does not use the assumption that the heuristic estimator is consistent. In their proof, it is sufficient that the heuristic estimator is optimistic. This motivates our idea that Nilsson [142] used the name A* for algorithms that use an optimistic heuristic estimator.
Gelperin [82] found flaws in the conditions for optimality in [103] and [104]. He refined some definitions and theorems in order to clarify the conditions for the optimality of \( A^* \), which actually results in prohibiting the use of common information in a better way than \( A^* \). Dechter and Pearl [46] point out that the main argument in the original proof, as outlined above, is especially weak on that particular assumption. It might happen that algorithm B gathers information while exploring a part of the graph unvisited by \( A^* \). It is not fair to call algorithm B more informed than \( A^* \), since the information gathered by B could have been found by \( A^* \) too. In [45], an actual example is given where \( A^* \) is surpassed (in terms of the number of nodes scanned) by an equally informed algorithm B. This example is based on a graph like the one given in Figure 5.1. The heuristic estimator provided is optimistic (but not consistent). The \( A^* \) algorithm will scan all nodes in the graph, as can easily be verified. Let us now assume that Algorithm B runs a Depth First Search on the graph, but will block parts of the graph that will surely not lead to a shorter path. Algorithm B scans node \( s \) and will label nodes 1 and 2. Due to the Depth First Search strategy, it will subsequently scan node 1 (\( A^* \) will choose node 2 as the second node to be scanned). From node 1, the destination node \( t \) is reached (\( d_t = 11 \)). During the scanning of node 1, algorithm B might set \( h_2 \leftarrow \max(h_2, h_1 - c_{12}) = 9 \), since algorithm B knows that \( h \) is optimistic (in Section 5.2.3 we will describe this update technique of \( h \) in more detail). In the next iteration, node 2 is selected. Since \( d_2 + h_2 = 2 + 9 = 11 = d_t \), algorithm B will not scan the outgoing arcs of node 2. Thus, nodes 3, \ldots, \( n \) are not scanned by algorithm B.

![Figure 5.1: A graph with an optimistic heuristic estimator for which \( A^* \) visits all nodes.](image)

As stated above, much research is done to prove the optimality of the \( A^* \) algorithm in a variety of problem instances. It is shown that defining optimality merely as being the number of nodes scanned, does not take into account the number of iterations needed. Martelli [132] proved that the use of a non-consistent optimistic heuristic estimator may lead to \( O(2^{|V|}) \) iterations. Furthermore, the algorithm that uses the smallest number of iterations, is not necessarily the fastest running algorithm (see for example [126]). In [150], research is done on other selection rules than \( d_v + h_v \), like \((1 - \alpha)d_v + \alpha h_v\), with \( 0 \leq \alpha < 1 \).
5.1 The family of $A^*$ algorithms

Holte [106] declared the optimality of $A^*$ over other equally informed algorithms to be one of the five common misconceptions concerning heuristic search. The research on the optimality of $A^*$ (in the sense of minimizing the number of nodes scanned) is not adequate. It lacks, for example, comparisons with bidirectional variants already known at that time.

In the remainder of this chapter, we will not focus on this so-called optimality of $A^*$. We instead show how heuristic search can be incorporated in various types of shortest path algorithms such as the ones described in Chapter 3. Furthermore, we will use heuristic estimates in bidirectional algorithms. Research on the use of heuristic estimates in bidirectional algorithms has a long history. We will summarize this history, before presenting a new theory that shows how heuristic estimates can be used in an efficient and straightforward way.

5.1.3 Commonly used consistent heuristic estimators

Assume that the coordinates $x_v$ of each node $v \in V$ are available. In many practical situations, the coordinates of a node are defined in $\mathbb{R}^2$. Nowadays, several large roadmaps are widely used. The coordinates of such a road map are defined as a latitude $\phi$ and a longitude $\lambda$ on the so-called WGS84-ellipsoid. In these cases, the coordinates $(\phi, \lambda)$ are first translated to geographical coordinates $(x, y, z)$ in $\mathbb{R}^3$.

In [103], two common heuristic estimators are described for graphs where the coordinates of each node are known and the arc costs represent either arc length or travel times. Their estimators are based on both the Euclidean-norm and the Manhattan-norm.

**Definition 5.10** A mapping $x \rightarrow ||x||$, with $x \in \mathbb{R}^n$ and $||x|| \in \mathbb{R}_+$, is called a norm if it satisfies the following conditions:

1. $||x|| \geq 0$.
2. $||a x|| = |a|||x||$, for all $a \in \mathbb{R}$.
3. $||x + y|| \leq ||x|| + ||y||$.

The $p$-norm of a vector $x = [x_1, x_2, \ldots, x_n]$ is defined as:

$$||x||_p = \left( \sum_{i=1}^{n} |x_i|^p \right)^{1/p}.$$ (5.5)

The $p$-norm is called the Manhattan-norm in case $p = 1$. If $p = 2$, it is called the Euclidean norm. A special case is the infinity-norm: $||x||_{\infty} = \max(|x_1|, |x_2|, \ldots, |x_n|)$. It is easy to see that:

$$\frac{||x||_1}{n} \leq ||x||_{\infty} \leq ||x||_2.$$ (5.6)
Definition 5.11 We call a strongly connected digraph $G = (V, A)$ a road map graph if the coordinates $x_v$ of each node $v \in V$ are available and if for any arc $(v, w) \in A$ the arc cost $c_{vw} \geq \alpha \|x_w - x_v\|_2$, for some constant $\alpha \in \mathbb{R}^{++}$.

Proposition 5.12 Let $G = (V, A)$ be a road map graph such that $c_{vw} \geq \alpha \|x_w - x_v\|_2$ for a given $\alpha \in \mathbb{R}^{++}$. Then, $d(v, w) \geq \alpha \|x_w - x_v\|_2$.

Proof: Let $P = (v_0 = v, (v_0, v_1), v_1, \ldots, (v_{k-1}, v_k), v_k = w)$ be a shortest $v$-$w$ path. We have:

$$d(v, w) = \sum_{i=0}^{k-1} c_{v_i, v_{i+1}} \geq \alpha \sum_{i=0}^{k-1} \|x_{v_{i+1}} - x_{v_i}\|_2 \geq \alpha \|x_w - x_v\|_2.$$ 

The last inequality follows from Definition 5.10.

For a road map graph, the mostly used heuristic estimator $h$ is based on the Euclidean distance to node $t$:

$$h_v = \alpha \|x_v - x_t\|_2.$$ 

(5.7)

By the triangle inequality, the heuristic estimator $h$, with $h_v$ as defined in (5.7), is consistent. If the arc costs $c_{vw}$ represent arc lengths, we have $\|x_v - x_w\|_2 \leq d(v, w)$ and thus $\alpha = 1$ can be used. If the arc costs $c_{vw}$ represent travel times, $\alpha$ can be set to $\frac{1}{\beta}$, where $\beta$ is the maximum speed over all arcs. If a graph is known to be a road map graph, the highest possible value of $\alpha$ can easily be determined during a preprocessing step [201]:

$$\alpha = \min_{(v, w) \in A} \frac{c_{vw}}{\|x_v - x_w\|_2}. $$

All illustrations in this chapter are based on this Euclidean heuristic estimator, unless stated otherwise. Figure 5.2 shows the search space of A* using an Euclidean estimator. This search space is much smaller than the search space of DIJKSTRA (see the left part of Figure 4.1).

Using a heuristic estimator $h$, the number of iterations needed to determine a shortest $s$-$t$ path can be reduced drastically. Yet the total running time can still be worse than the case where no heuristic estimator is used, since in each iteration the value of $h_v$ has to be evaluated, which might require considerable computational effort. If for example the computation effort of the Euclidean distance (5.7)) is significant due to the square root operation, one could consider the use of a heuristic estimator, like:

$$h_v = \frac{\alpha \|x_v - x_t\|_1}{n}. $$

(5.8)
5.1 The family of A∗ algorithms

The heuristic estimator, defined by (5.7), results in lower bounds at least as tight as the heuristic estimator defined by (5.8). Despite this, the A∗ algorithm based on the heuristic estimator (5.8) might run faster as a result of lower computational costs for evaluating the heuristic estimator during execution.

For many years, the Euclidean heuristic estimator \( h \) (5.7) has commonly been used for road map graphs. We will see in Section 6.3.3 that in 2004, Goldberg and Harrelson [89] introduced an improved heuristic estimator based on certain pre-calculated shortest path trees instead of on additional information gathered from node coordinates.

5.1.4 Equivalency between Dijkstra and A∗

It is obvious that the A∗ algorithm, using the consistent heuristic estimator \( h \) with \( h_v = 0 \) for all \( v \in V \), is equivalent to Dijkstra. Martelli [132] proved that both algorithms are equivalent for any consistent heuristic estimator \( h \).

**Theorem 5.13** Dijkstra (Algorithm 3.6), using the modified arc costs \( c'_{vw} = c_{vw} + h_w - h_v \), is equivalent to the A∗ algorithm (Algorithm 5.1) using the original arc costs \( c_{vw} \) and the consistent heuristic estimator \( h \).

**Proof:** The heuristic estimator \( h \) is consistent, so \( h_v - h_w \leq d(v, w) \leq c_{vw} \). Hence, \( c'_{vw} \geq 0 \), and therefore these modified arc costs can be used Dijkstra. Following Theorem 3.1, a label \( d_D^D \ < \infty \) in Dijkstra is the length of some \( s-v \) path \( P \):

\[
d_D^D = \sum_{(u, w) \in P} c'_{uw} = \sum_{(u, w) \in P} c_{uw} + h_v - h_s.
\]

(5.9)

**Figure 5.2:** A∗ using an Euclidean heuristic estimator.
Notice that $h_s$ is constant and $\sum_{(u,w) \in P} c_{uw}$ denotes the label $d^*_v$ of node $v$ in the $A^*$ algorithm. Selecting a pivot node $v = \arg \min \{ d^D_v | v \in Q \}$ in Dijkstra is therefore equivalent to selecting a pivot node $v = \arg \min \{ d^*_v + h_v | v \in Q \}$ in the $A^*$ algorithm.

To complete the proof, we show that the Label-Update procedure in Dijkstra, which uses the modified arc costs, is equivalent to the Label-Update procedure in the $A^*$ algorithm (using the original arc costs). The check performed in this procedure is $d^D_v + c'_{vw} < d^D_w$. By (5.9), we have:

$$d^*_v + h_v - h_s + c_{vw} + h_w - h_v = d^*_v - h_s + c_{vw} + h_w < d^*_w + h_w - h_v.$$  

Hence, the check performed in Dijkstra is $d^*_v + c_{vw} < d^*_w$, which is the same as is performed in the $A^*$ algorithm.

Thus, the worst case complexity of the $A^*$ algorithm is the same as the worst case complexity of Dijkstra.

### 5.2 Label-correcting algorithms and heuristic estimators

In this section we show how heuristic estimators can be incorporated in label-correcting algorithms (see Section 3.3). Recall that label-correcting algorithms distinguish themselves from label-setting algorithms by the possibility that a node re-enters the candidate set $Q$. Thus, label-correcting algorithms usually need more iterations than label-setting algorithms. However, label-correcting algorithms can outperform label-setting algorithms in terms of running time, since a label-correcting algorithm usually selects a pivot node in $O(1)$ time.

The $A^*$ algorithm (Algorithm 5.1), which uses an optimistic (but inconsistent) heuristic estimator, is in fact a label-correcting algorithm. In [132], a graph family is presented with an optimistic heuristic estimator for which $A^*$ is clearly label-correcting. Figure 5.3 shows graph $G_3$ of this family. It is clear that $h$ is an optimistic, yet inconsistent estimator. $A^*$ will scan node 2 twice, which demonstrates that $A^*$ is label-correcting on this graph.

Although $A^*$, using an inconsistent heuristic estimator, is a label-correcting algorithm, it does not profit from the $O(1)$ time needed to select a pivot node. The question naturally arises whether or not the use of inconsistent heuristic estimators can be profitable. We will answer this question in Section 5.2.2.
5.2 Label-correcting algorithms and heuristic estimators

5.2.1 Adding heuristic estimators to label-correcting algorithms

Let us first focus on the label-correcting algorithms mentioned in Section 3.3. Especially the potential-invariant algorithms (Section 3.3.1), like Bellman-Ford-Moore, that do not use any actual node label while choosing a pivot node, are unlikely to profit from a heuristic estimator.

However, as soon as the target node \( t \) gets a finite label, we can use a sharper stop condition (compared to the one provided in Section 3.3.4) if \( h \) is optimistic [121].

**Proposition 5.14** Let \( h \) be an optimistic heuristic estimator. Let \( Q \) be the candidate set of a label-correcting algorithm. We may remove any node \( v \in Q \) for which \( d_v + h_v \geq d_t \), thereby omit scanning its outgoing arcs.

**Proof**: We distinguish two situations:

(i) \( d_v = d(s, v) \).

(ii) \( d_v > d(s, v) \).

In case (i), \( d_v \) equals \( d(s, v) \). With \( h_v \leq d(v, t) \), \( d(s, v) + d(v, t) \geq d_v + h_v \geq d_t \geq d(s, t) \). Thus, any shortest path from \( s \) to \( t \) via node \( v \) will not be shorter than the \( s-t \) path already found. In case (ii), it is needless to scan node \( v \), since \( d_v \) is not a sharp label. If \( v \) is positioned on the shortest \( s-t \) path, its label will be updated in a later iteration.

In [19], a similar possibility is mentioned to tighten the test for entering nodes to \( Q \); only add a node \( v \) to \( Q \) for which \( d_v + c_{vw} < \min (d_w, d_t - h_w) \).

Furthermore, the SLF strategy (see page 49) can be improved by adding the estimate to the label values when comparing the label value of the entering node \( v \) with the top node \( w \) of \( Q \); if \( d_v + h_v \leq d_w + h_w \), \( v \) is entered at the top of \( Q \), otherwise, \( v \) is entered at the bottom of \( Q \). With this strategy, the minimum label selection policy of the \( A^* \) algorithm is approximately emulated.
The same applies for the LLL strategy (see page 54). At each iteration, when node \( v \) at the top of \( Q \) has a label \( d_v \) such that
\[
d_v + h_w > \frac{\sum_{w \in Q} d_w + h_w}{|Q|},
\]
then node \( v \) is not removed from \( Q \), but is instead repositioned at the bottom of \( Q \).

### 5.2.2 Inconsistent heuristic estimators

As stated in Definition 5.5, a heuristic estimator is called consistent if \( h_v - h_w \leq d(v, w) \) for all pairs of nodes \( v, w \in V \). If, for a heuristic estimator \( h \), there exists at least one pair of nodes \( v, w \in V \) for which \( h_v - h_w > d(v, w) \), then \( h \) is called inconsistent. In [204], it is stated that the term ‘inconsistent heuristic estimator’ sounds like something that has to be avoided. Martelli [132] showed that the use of an inconsistent estimator can lead to poor performance of the \( A^* \) algorithm. By Theorem 5.7, we know that the use of a consistent heuristic estimator results in a label-setting algorithm that executes \( O(|V|) \) iterations. Thus, it is not strange that in most research the use of consistent heuristic estimators is preferred.

Furthermore, some researchers [127, 169] believe that most optimistic heuristic estimators are consistent. The idea is that it is difficult to construct an optimistic heuristic estimator that is inconsistent. However, in [204] and [206], it is shown that it is easy to construct an optimistic yet inconsistent heuristic estimator and that the use of such a heuristic might even be preferable. We will highlight two of these inconsistent heuristic estimators, since they can easily be used in the next chapter.

In many practical situations, a set \( H \) of heuristic estimators is used. To overcome the possible pitfalls of one specific heuristic estimator in \( H \), the following heuristic estimator is used:
\[
h^*_v = \max_{h \in H} h_v.
\]

By Theorem 5.8, we know that, if all heuristic estimators in \( H \) are consistent, \( h^* \) is consistent. To determine \( h^* \), all members of \( H \) have to be evaluated. The computational effort for determining \( h^* \) might have such a negative effect on the total running time of the \( A^* \) algorithm, that it is not compensated by the possibly diminishing number of iterations performed by the algorithm. It might be better to randomly select for each node a particular member of \( H \). For most implementations, it is necessary to keep the relation between the node and the selected heuristic estimator fixed during the process. This can easily be established by storing the value \( h_v \) for node \( v \) the first time it is evaluated. Assigning a random member of \( H \) to each node then may easily result in an inconsistent heuristic estimator.

Another ‘natural’ way of creating an inconsistent estimator is based on incomplete heuristic estimators. If, for some nodes, there is no heuristic estimate possible,
the heuristic estimate for those nodes can be considered zero. This results in an optimistic yet inconsistent heuristic estimator function.

As we will see in Chapter 6, the use of preprocessing techniques will lead to the possible use of both types of inconsistent heuristic estimators.

### 5.2.3 Variable heuristic estimates

Before we will actually introduce variable heuristic estimates, we define dominance of one heuristic estimator above the another.

**Definition 5.15** Let $h'$ and $h''$ be two optimistic heuristic estimators. If $h''_v \geq h'_v$ for all $v \in \mathcal{V}$ and $h''_v > h'_v$ for at least one $v \in \mathcal{V}$, then $h''$ dominates $h'$.

Intuitively, one might expect that $A^*$ using $h''$ will scan less nodes than $A^*$ using $h'$. In [169], this is motivated as follows: From the proof of Theorem 5.4 it is clear that, upon termination of the $A^*$ algorithm (using an optimistic heuristic estimator $h$) all nodes $v \in \mathcal{V}$ for which $d_v + h_v < d(s,t)$ are scanned. Thus, all nodes $v \in \mathcal{V}$, for which $h_v < d(s,t) - d_v$ are scanned. Since $h''$ dominates $h'$, $A^*$ using $h''$ will never expand more nodes than $A^*$ using $h'$, except for some nodes $w$ for which $d_w + h_w = d(s,t)$. In [106], a pathological graph is presented where $A^*$, using a dominated heuristic estimator, scans less nodes than $A^*$ using a dominant heuristic estimator. For any extra node $w$, scanned by the latter variant of $A^*$, indeed the equality $d_w + h_w = d(s,t)$ holds. Since this graph is pathological however, in most situations a dominant heuristic estimator is preferred.

Méró, [134] considers the estimate $h_v$ as a variable rather than as a constant. As soon as node $v$ is selected as pivot node, the following updates are performed:

$$h_w \leftarrow \max (h_w, h_v - c_{vw}) \quad \forall (v,w) \in \delta^+(v). \quad (5.10)$$

Estimate $h_v$ of the pivot node $v$ might be updated too:

$$h_v \leftarrow \max \left( h_v, \min_{(v,w) \in \delta^+(v)} (c_{vw} + h_w) \right). \quad (5.11)$$

The original proposal of Méró for updating $h_v$ of the pivot node $v$, was incorrect. Equation (5.11) is the corrected version that can be found in [206].

**Proposition 5.16** If, during the execution of the $A^*$ algorithm, the heuristic estimates are updated in each iteration using the statements (5.10) and (5.11), then for any node $v \in \mathcal{V}$, one has $h_v \leq d(v,t)$, provided that the initial heuristic estimates were optimistic.

**Proof:** We will prove by induction that, at the beginning of each iteration $h_v \leq d(v,t)$, for all nodes $v \in \mathcal{V}$. It is clearly true at the beginning of the first iteration,
since the provided heuristic estimator $h$ is optimistic. Now, suppose it is true at the
beginning of another iteration where node $v$ is selected as pivot node. By definition,
we have:

$$d(v, t) = \min_{(v, w) \in \delta^+(v)} (c_{vw} + d(w, t)). \quad (5.12)$$

If $h_w$ is updated according to (5.10), we have $h_w = h_v - c_{vw}$. From the induction
hypothesis, we have $h_w \leq d(v, t) - c_{vw}$. It follows from (5.12) that $d(v, t) \leq d(w, t) + c_{vw}$. Thus, $h_w \leq d(w, t)$. If $h_v$ is updated during the iteration according to (5.11),
we have $h_v = \min_{(v, w) \in \delta^+(v)} (h_w + c_{vw})$. From (5.12) and the induction hypothesis
we have:

$$d(v, t) = \min_{(v, w) \in \delta^+(v)} (c_{vw} + d(w, t)) \geq \min_{(v, w) \in \delta^+(v)} (c_{vw} + h_w) = h_v.$$

Thus, at the end of each iteration we have $h_v \leq d(v, t)$ for all $v \in V$. This completes
the induction proof.

Proposition 5.17 Let $h$ be an optimistic heuristic estimator and let these estimates
be modified according to (5.10) and (5.11) during the execution of $A^\ast$. Then, the
$A^\ast$ algorithm terminates and, upon termination, $d_t$ is the length of a shortest $s$-$t$
path.

Proof: The proof of Theorem 5.4 about the finiteness and correctness of $A^\ast$ never
makes use of the implicit assumption that the heuristic estimates are constant. These
estimates only have to satisfy $h_v \leq d(v, t)$ during the process. Since by Propo-
sition 5.16 this is still the case if the provided optimistic estimates are modified
according to (5.10) and (5.11), we conclude that $A^\ast$, using the described modific-
tions on the heuristic estimates, is finite and correct.

Modifying the heuristic estimates as described above is nowadays referred to as the
Pathmax procedure. Suppose the initial heuristic estimator is consistent. Then
$h_v - h_w \leq c_{vw}$ for all $(v, w) \in A$. It is immediately clear that no estimate will be
modified by (5.10). Although the estimate $h_v$ of the pivot node $v$ might be modified
by (5.11) this does not make any sense. By Theorem 5.7, we know that $A^\ast$ with a
consistent heuristic estimator is a label-setting algorithm. Thus, node $v$ will never
re-enter the candidate set $Q$. The possibly improved value of $h_v$ will never be used.

Therefore, Pathmax is normally used with inconsistent heuristic estimates. Al-
though Pathmax can lead to a very substantial reduction in the number of itera-
tions, it still might happen that a node is selected more than once as a pivot node.
Thus, using $A^\ast$ with an inconsistent heuristic estimator and Pathmax, still results
in a label-correcting algorithm [143].
5.3 Bidirectional algorithms and heuristic estimators

It is possible to incorporate heuristic estimators in a bidirectional algorithm. Bidirectional search can be implemented using symmetric heuristic functions, where both the forward and backward processes estimate the distance between a node and the target in a similar way. Another type is the heuristic that is called balanced in this thesis. This type of heuristic has been introduced in [112] and incorporated in several published experiments with large-scale actual road maps ([89, 121]). According to these papers, the balanced heuristic is beneficial because the post-phase is very short. In this section, we will show how the benefits from a balanced approach can be carried over to the symmetric approach.

In this section, we use $h^s$ to denote a consistent heuristic estimator towards $t$ and $h^t$ to denote a consistent heuristic estimator towards $s$.

5.3.1 The symmetric approach

A general framework to use heuristic estimators in a bidirectional shortest path algorithm is presented in Algorithm 5.2 and Algorithm 5.3. These algorithms are closely related to Algorithms 4.1 and 4.2. The Iterate* procedure differs from the Iterate procedure in the way the pivot node is selected (line 1).

Algorithm 5.2: Iterate*(G, Q, h) (LABEL-SETTING)

1: Let $v \leftarrow \text{arg min } \{d_v + h_v | v \in Q\}$
2: $Q \leftarrow Q - \{v\}$
3: if $v \in \tilde{R}$ then
4:     if $u = \text{NIL}$ or $d^a_v + d^f_v < d^a_u + d^f_u$ then
5:         $u \leftarrow v$
6:     else
7:     foreach $(v, w) \in \delta^+(v)$ do
8:         LABEL-UPDATE($v, w, Q$)

Algorithm 5.3 uses $h^s$ and $h^t$ bidirectionally.

In Section 4.2.2, several selection rules are described to decide whether a particular iteration of Algorithm 4.2 will be a forward or a backward iteration. These rules can also be applied to Algorithm 5.3.

The most challenging part in Algorithm 5.3 is when a correct and efficient stop condition is used. Pohl [158] suggested the following stop condition, which can be used as soon as at least one meeting node $u$ is found:
Similarly, if \( u \) is an end node, then we have \( d_u^s + d_u^t = d_u^s + d_u^t = d(s, t) \).

Finally, Algorithm 5.3 will always terminate since \( s \) and \( t \) are both possible meeting nodes for which stop condition (5.13) holds.

It might happen that \( u \), at the end of Algorithm 5.3, equals either \( s \) or \( t \). Typically, both search spaces (forward and backward) might have a severely large intersection. To reduce the amount of work, Kwa [128] introduced several concepts such as nipping and pruning. As soon as a pivot node \( v \) is selected in the primary search direction and \( v \in \tilde{R} \), the pivot node \( v \) enters \( \tilde{R} \) without scanning its outgoing arcs.

---

Algorithm 5.3: Bidirectional A*(\( G^s, G^t, s, t, h^s, h^t \))

```
1: \( u \leftarrow \text{NIL} \)
2: \text{INITIALIZE-SINGLE-SOURCE}(G^s, s, Q^s)
3: \text{INITIALIZE-SINGLE-SOURCE}(G^t, t, Q^t)
4: \text{ITERATE}^s(G^s, Q^s, h^s)
5: \text{ITERATE}^t(G^t, Q^t, h^t)
6: \text{repeat}
7: \quad \text{if a certain selection rule holds then}
8: \quad \quad \text{ITERATE}(G^s, Q^s, h^s)
9: \quad \text{else}
10: \quad \quad \text{ITERATE}(G^t, Q^t, h^t)
11: \text{until a certain stop condition holds}
```

\[
d_u^s + d_u^t \leq \max \left\{ \min_{v \in Q^s} d_v^s + h_v^s - h_t^s, \min_{v \in Q^t} d_v^t + h_v^t - h_s^t \right\}. \quad (5.13)
\]

**Theorem 5.18** [158] Let \( G \) be a strongly connected graph with nonnegative arc lengths. Let \( h^s \) and \( h^t \) be consistent heuristic estimators towards \( t \) and \( s \), respectively. Algorithm 5.3, using stop condition (5.13), will then terminate and upon termination \( d_u^s + d_u^t \) is the length of a shortest \( s \)-\( t \) path.

**Proof:** From Proposition 4.1, we know that at least one meeting node \( u \) will be found during the execution of Algorithm 5.3. From Theorem 5.7, we find for any meeting node \( u \) that the length of the shortest path from \( s \) to \( t \) via \( u \) has length \( d_u^s + d_u^t \). During the execution of Algorithm 5.3, several meeting nodes might be found. Node \( u \) always points to a meeting node for which \( d_u^s + d_u^t \) is minimal.

Now, suppose Algorithm 5.3 terminates. If \( u = t \), we have \( d_u^s + d_u^t = d_s^s + d_t^t = d(s, t) \). Similarly, if \( u = s \), we have \( d_u^s + d_u^t = d_s^s + d_t^t = d(s, t) \). If \( u \neq s \) and \( u \neq t \), we complete the proof of correctness by contradiction. Suppose there exists a path \( P = (s = v_0, (v_0, v_1), v_1, \ldots, v_k-1, (v_k-1, v_k), v_k = t) \), with \( \ell(P) < d_u^s + d_u^t \). Then, by induction, \( v_i \in Q^s, v_{i-1} \in \mathcal{R}^s \) and \( v_j \in Q^t, v_{j+1} \in \mathcal{R}^t \), with \( 1 \leq i \leq j \leq k-1 \) exists, otherwise a meeting node \( u \) on \( P \) would have been found. But \( d_v^s + h_v^s - h_t^s \leq \ell(P) < d_u^s + d_u^t \) and \( d_v^t + h_v^t - h_s^t \leq \ell(P) < d_u^s + d_u^t \). The stop condition (5.13) is therefore not met and the algorithm will not be stopped.

Finally, Algorithm 5.3 will always terminate since \( s \) and \( t \) are both possible meeting nodes for which stop condition (5.13) holds. \( \square \)
The idea is that the opposite process already determined the shortest path from node \( v \) to the primary destination. Not scanning the outgoing arcs is called *nipping*. As soon as a node \( v \) is nipped in the primary process, all nodes \( w \in \bar{Q} \) with node \( v \) as a direct or indirect predecessor need not to be scanned anymore in the opposite process. Those nodes might be removed from \( \bar{Q} \). This process is called *pruning* or *retrospective nipping*.

In Figure 5.4, the positive effect of nipping and pruning is visualized compared to Pohl’s stop condition. From the left part of the figure it is obvious that, without nipping and pruning, both search spaces have a large intersection. The right part of the picture still shows a long borderline between both search spaces. Such a long borderline between both search spaces is very common when so-called symmetric heuristic estimators are used.

In [116], it is stated that the major effort of Algorithm 5.3 is spent after a meeting node is found. In Figure 5.5 one can see the search space as soon as a meeting node is found. The search spaces of parts (a) and (b) are clearly much larger in Figure 5.8. Therefore, most of the related literature [89, 96, 112, 121] shows preference for the use of so-called balanced heuristic estimators over symmetric heuristic estimators.

5.3.2 The balanced approach

Ikeda [112] exploited the equivalence between the A* algorithm with a consistent heuristic, and Dijkstra (see Section 5.1.4). Running the A* algorithm on a graph with arc costs \( c_{vw} \) and a consistent heuristic estimator \( h \) is equivalent to Dijkstra on a graph with modified arc costs:
In terms of the bidirectional label-setting algorithm (Algorithm 4.2), where the modified arc costs \( c'_{vw} \) are used in the primary process, the opposite process should have the same arc costs but reversed. So every \( c'_{vw} \) used in the primary process should correspond to a value \( \tilde{c}'_{wv} \) on the opposite side. The following equality holds:

\[
\begin{align*}
c'_{vw} &= \tilde{c}'_{wv} = \tilde{c}_{wv} + \tilde{h}_v - \tilde{h}_w = c_{vw} + \tilde{h}_v - \tilde{h}_w. 
\end{align*}
\]  

(5.15)

Comparing (5.14) and (5.15), we see that we are allowed to transform bidirectional A* into bidirectional Dijkstra only if the two estimators satisfy \( \tilde{h}_v = -h_v \), for all \( v \in V \). A heuristic estimator \( h \) with this property is called balanced\(^1\).

In [112], the estimate \( (h^s_v - h^t_v)/2 \) is used in the forward search and the estimate \( (h^t_v - h^s_v)/2 \) in the backward search. It is easy to see (if \( h^s_v \geq 0 \) and \( h^t_v \geq 0 \)) that:

\[
\begin{align*}
\frac{1}{2}(h^s_v - h^t_v) &\leq h^s_v, \\
\frac{1}{2}(h^t_v - h^s_v) &\leq h^t_v.
\end{align*}
\]

These inequalities show that new estimators are inferior to the original estimators \( h^s \) and \( h^t \). But the new estimators are balanced. Hence, we can incorporate heuristic search in Algorithm 4.2 by using the modified arc costs:

\(^1\)This type of heuristic was confusingly called ‘consistent’ in [89]. In related literature, the term consistent is mostly used in the same sense as we do here.
5.3 Bidirectional algorithms and heuristic estimators

\[ c_{vw}^s = c_{vw}^s + \frac{1}{2}(h_w^s - h_v^l) - \frac{1}{2}(h_v^s - h_v^l) \]
\[ c_{vw}^t = c_{vw}^t + \frac{1}{2}(h_v^t - h_v^s) - \frac{1}{2}(h_w^t - h_w^s) = c_{vw}^s. \]

From Section 4.2.1, we know that running the bidirectional algorithm (Algorithm 4.2) on a strongly connected graph will eventually lead to at least one meeting node \( u \). As soon as a meeting node \( u \) is determined, we run a post-phase (Algorithm 4.3). At the end of the post-phase, node \( u \) surely lies on a shortest \( s-t \) path. We have:

\[ d_u^s + d_u^t = \sum_{(v,w) \in P_u} c_{vw}^s + \sum_{(v,w) \in P_u} c_{vw}^t = \sum_{(v,w) \in P_u} c_{vw}^s + \frac{1}{2}(h_w^s - h_v^l) - \frac{1}{2}(h_v^s - h_v^l) + \]
\[ \sum_{(v,w) \in P_u} c_{vw}^t + \frac{1}{2}(h_v^t - h_v^s) - \frac{1}{2}(h_w^t - h_w^s) = \]
\[ \sum_{(v,w) \in P_u} c_{vw}^s + \sum_{(v,w) \in P_u} c_{vw}^t - \frac{1}{2}(h_w^s - h_v^t) - \frac{1}{2}(h_v^t - h_w^s). \]

So \( d_u^s + d_u^t \) denotes the length of a shortest \( s-t \) path, shifted by the constant \( \frac{1}{2}(h_v^s - h_v^l) + \frac{1}{2}(h_w^t - h_w^s) \).

The strength of a balanced approach is that we can stop the bidirectional algorithm as soon as a meeting node \( u \) is found, after which we only have to run a relatively short post-phase. The weakness of a balanced approach is that we cannot use the full strength of the original estimators \( h^s \) and \( h^t \).

Figure 5.6 shows the search space of a bidirectional A* algorithm, using balanced heuristic estimators. The resulting search space is clearly smaller than the ones showed in Figure 5.4. The popularity of such a balanced approach is obvious. In the next sections, we will provide some new theory in favor of symmetric heuristic estimators.

5.3.3 A new symmetric approach

Recall that in the symmetric approach, a lot of work still has to be done after a meeting node \( u \) is found [116]. In [154, 155] we showed a way to drastically shorten this ‘post-process’ for bidirectional label-setting algorithms. The main idea is to keep track of \( f = \max_{v \in R} d_v + h_v \). This can simply be done by adding \( f \leftarrow d_v + h_v \) directly after a pivot node \( v \) is selected in Algorithm 5.2.

Our algorithm runs Algorithm 5.4 in combination with the Iterate* procedure, as stated in Algorithm 5.5.

**Proposition 5.19** At the end of each iteration of Algorithm 5.4, using the consistent estimators \( h^s \) and \( h^t \), the following conditions hold: \( d^s_v + h^s_v \leq f^s \leq d^s_w + h^s_w \) for all \( v \in R^s \) and \( w \in Q^s \), and \( d^t_v + h^t_v \leq f^t \leq d^t_w + h^t_w \) for all \( v \in R^t \) and \( w \in Q^t \).
Figure 5.6: Bidirectional $A^*$ using balanced heuristic estimators.

Algorithm 5.4: New Bidirectional $A^*(G^*, G^t, s, t, h^s, h^t)$

1: $u \leftarrow \text{NIL}$
2: Initialize-Single-Source($G^*, s, Q^s$)
3: Initialize-Single-Source($G^t, t, Q^t$)
4: New Iterate*($G^*, Q^s, h^s, t$)
5: New Iterate*($G^t, Q^t, h^t, s$)
6: repeat
7: if a certain selection rule holds then
8: New Iterate*($G^*, Q^s, h^s$)
9: else
10: New Iterate*($G^t, Q^t, h^t$)
11: until $Q^s = \emptyset$ or $Q^t = \emptyset$

Algorithm 5.5: New Iterate*($G, Q, h, t$) (LABEL-SETTING)

1: Let $v \leftarrow \text{arg min} \{d_v + h_v | v \in Q\}$
2: $Q \leftarrow Q - \{v\}$
3: $f \leftarrow d_v + h_v$
4: if $u = \text{NIL}$ or $\max\{d_v + \bar{f} - \bar{h}_v, d_v + h_v - h_t\} < d_u + \bar{d}_u$ then
5: foreach $(v, w) \in \delta^+(v)$ do
6: Label-Update($v, w, Q$)
7: if $w \in \bar{R}$ then
8: if $u = \text{NIL}$ or $d_w + \bar{d}_w < d_u + \bar{d}_u$ then $u \leftarrow w$
Proof: We prove this proposition by induction on the iterations of the forward search. The proof for the backward search is similar. At the end of the first iteration, we have $\mathcal{R} = \{s\}$ and $\mathcal{Q} = \{w|(s, w) \in \delta^+(s)\}$. Since $f = d^+_w + h^+_w$, the inequality $d^+_w + h^+_w \leq f^*$ for all $v \in \mathcal{R}^s$ holds at the end of the first iteration. For node $w \in \mathcal{Q}$ we have:

$$d^+_w + h^+_w = d^+_w + c_{sv} + h^+_w \geq d^+_w + h^+_w = f.$$  

The inequality holds, due to the consistency of $h^*$ (see Proposition 5.6). The proposition holds at the end of the first iteration. Now, suppose the proposition holds at the end of a certain iteration. In the next iteration, pivot node $v$ is selected. At that moment, we have:

$$d^+_v + h^+_v \leq d^+_w + h^+_w, \ \forall w \in \mathcal{Q}. \quad (5.16)$$

According to the induction hypothesis, we have: $d^+_v + h^+_v \geq f \geq d^+_w + h^+_w$ for all $u \in \mathcal{R}$. At the end of the iteration, $f$ is updated ($f = d^+_v + h^+_v$) and only node $v$ has entered $\mathcal{R}$. Thus, the first inequality of the proposition holds at the end of this iteration.

For the second inequality, we have (5.16). During the iteration, $d^+_w$ can be decreased only for nodes $w$ adjacent to node $v$. If $d^+_w$ is decreased, we have:

$$d^+_w + h^+_w = d^+_w + c_{vw} + h^+_w \geq d^+_v + h^+_v = f.$$  

Again, the inequality holds due to the consistency of $h^*$. This completes the induction proof. \qed

Proposition 5.20 At the beginning of each iteration of Algorithm 5.4, assuming that $h^*$ and $h^1$ are consistent estimators, the following inequalities hold: $d(s, v) \geq f^* - h^*_v$ for all $v \notin \mathcal{R}^s$, and $d(v, t) \geq f^1 - h^1_v$ for all $v \notin \mathcal{R}^t$.

Proof: We prove $d(s, v) \geq f^* - h^*_v$ for all $v \notin \mathcal{R}^s$. The proof for the second inequality is similar. Let $v \notin \mathcal{R}^s$ and $P = (s = v_0, v_1, v_2, \ldots, v_{k-1}, v_k = v)$ a shortest $s$-$v$ path be given. Following Theorem 5.2, we have at least one node $v_i$ $0 \leq i \leq k$ on $P$ with $v_i \in \mathcal{Q}^s$ and $d(s, v_i) = d^s_{v_i}$. For such a node $v_i$, the following condition holds:

$$d(s, v) = d(s, v_i) + d(v_i, v) = d^s_{v_i} + d(v_i, v) \geq d^s_{v_i} + h^s_{v_i} \geq f^* - h^*_v. \quad (5.17)$$

The first inequality comes from the consistency of the estimator $h^*$. The second inequality holds due to Proposition 5.19, since $v_i \in \mathcal{Q}^s$. \qed

In line 4 of the New Iterate* procedure (Algorithm 5.5), we use (5.17) to skip the scanning of the outgoing arcs of a node $v$, for which either $d_v + h_v$ or $d_v + f - h_v$ is
at least as large as the length of shortest s-t path found so far. Both \(d_v + h_v\) and \(d_v + \hat{f} - \hat{h}_v\) are lower bounds for the length of any shortest s-t path via node \(v\).

The new lower bound is highly effective, as can be seen in Figure 5.8 part (c). The search space is now comparable to the search space showed in Figure 5.6, where balanced heuristic estimators are used. Thus, it might be preferable to use a symmetric approach, since the evaluation of symmetric heuristic estimator functions is faster.

The only restriction on the estimators \(h^s\) and \(h^t\) is that both estimators have to be consistent. So it is possible to use the new lower bound also with consistent balanced estimators. Recall that \(h^s\) and \(h^t\) are balanced if \(h^s_v = -h^t_v\) for all \(v \in \mathcal{V}\).

**Proposition 5.21** Running Algorithm 5.4 with the consistent balanced estimators \(h^s\) and \(h^t\), is equivalent to the balanced approach that is described in Section 5.3.2. In particular, as soon as a meeting node \(u \in R^s \cap R^t\) is found, outgoing arcs of other nodes are not scanned anymore.

**Proof:** As soon as a meeting node \(u \in \mathcal{Q}\) is found, the length of any shortest s-t path via node \(u\) is \(d^u_v + d^u_w\). Assume that a forward search iteration is performed (the proof for a backward search iteration is similar). Let \(v \in \mathcal{Q}\) be the selected pivot node. By Proposition 5.20, we have \(d(s, v) \geq f^s - h^s_v\):

\[
\begin{align*}
d(s, v) + f^t - h^t_v &\geq f^s - h^s_v + f^t - h^t_v = f^s + f^t \geq d^u_v + d^u_w.
\end{align*}
\]

The last inequality comes from Proposition 5.19. We conclude that the outgoing arcs of node \(v\) are not scanned. \(\square\)

Recently, Rios and Chaimowicz [164] successfully implemented the lower bound described in [154], in a parallel new bidirectional A* (PNBA*) algorithm. They tested their algorithm on several problem domains: the fifteen puzzle (see [182]) and grid pathfinding on randomly generated mazes with uniform and non-uniform costs. Their results showed a significant reduction in the running times.

The quality of the new lower bound depends highly on the value \(f\). The value \(f\) is a lower bound for \(d_v + h_v\), for all \(v \in \mathcal{Q}\). In a label-setting algorithm, the \(f\) value can easily be determined directly after selecting a pivot node (see Algorithm 5.5). Recall that the difference between label-setting algorithms and label-correcting algorithms is that in a label-correcting algorithm, any node can be selected as a pivot node. For a selected pivot node \(v\) in a label-correcting algorithm, it is likely that \(d_v + h_v \neq \min\{d_w + h_w|w \in \mathcal{Q}\}\). Thus, in a label-correcting algorithm, \(f\) cannot be determined without some additional computational effort. The only restriction on \(f\) is that it has to be a lower bound for \(d_v + h_v\) for all \(v \in \mathcal{Q}\). Therefore, it is possible to determine the \(f\) value no earlier than the moment where a meeting node is found. From that moment on, \(f\) can be kept as a lower bound or it can be improved by redetermining it after every \(n\) iterations. However, in our experiments we were not able to find a (generic) strategy for updating \(f\) in a label-correcting algorithm such
5.3 Bidirectional algorithms and heuristic estimators

that the algorithm performs better than its variant, which uses balanced heuristic estimators.

In the next section, we describe another method to implement symmetric heuristic estimators effectively, even for label-correcting algorithms.

5.3.4 Divide and conquer

Whangbo [197] proposed a method that significantly reduces the work to ensure that a shortest path is found, once a meeting node \( u \) is found. His method is suitable for road map graphs (see Definition 5.11). Algorithm 5.6 reflects his ideas\(^2\).

**Algorithm 5.6: Bidirectional A* (scalar) (\( G^s, G^t, s, t, h^s, h^t \))**

1: \( u \leftarrow \text{NIL} \)
2: \( u_0 \leftarrow \text{NIL} \)
3: \( \text{Initialize-Single-Source}(G^s, s, Q^s) \)
4: \( \text{Initialize-Single-Source}(G^t, t, Q^t) \)
5: repeat
6: if a certain selection rule holds then
7: \( \text{Iterate (scalar projection)}(G^s, Q^s, h^s) \)
8: else
9: \( \text{Iterate (scalar projection)}(G^t, Q^t, h^t) \)
10: until \( Q^s = \emptyset \land Q^t = \emptyset \)

If we compare this algorithm with Bidirectional A* (Algorithm 5.3), we see the following differences: First, we have a node \( u_0 \) (initialized to NIL) besides node \( u \). A node \( u \) is called a 'meeting node' if \( d^s_u < \infty \) and \( d^t_u < \infty \). As soon as the first meeting node \( u \) is found, we set \( u_0 \leftarrow u \) and we use \( u_0 \) to divide the search space to be explored over the forward and the backward search. As can be seen in Algorithm 5.7, \( u \) can be updated as soon as a shorter path via a new meeting node is found. It is essential that \( u_0 \) is not updated and points to the first meeting node found.

Let us assume that the check, performed in line 9 of Algorithm 5.7, always holds. Later, we will use this check to divide the search space.

Another difference between Algorithms 5.6 and 5.3 is based on the definition of a meeting node. In Algorithm 5.3, a node \( v \) is a meeting node if \( v \in R^s \cup R^t \). Thus, besides having labels \( d^s_v < \infty \) and \( d^t_v < \infty \), the additional properties \( v \notin Q^s \) and \( v \notin Q^t \) are required. To ensure that at least one meeting node is found eventually, Algorithm 5.3 needs an initial call to the iteration procedures for both directions. In Algorithm 5.6, these initial iterations are not needed anymore. The two calls to the Initialize-Single-Source procedure ensure that at least one meeting node will be found.

\(^2\)Since we found an error in the main proof of Whangbo’s paper, we present a new theory here, to prove his concept.
Algorithm 5.7: Iterate (Scalar Projection)\((G, Q, h)\)

1: Select a pivot node \(v \in Q\)
2: \(Q \leftarrow Q - \{v\}\)
3: if \(\tilde{d}_v < \infty\) then
4: if \(u = \text{NIL}\) then
5: \(u \leftarrow v\)
6: \(u_0 \leftarrow u\)
7: else if \(d_s^u + d_t^v < d_s^v + d_t^u\) then
8: \(u \leftarrow v\)
9: if a certain check holds then
10: foreach \((v, w) \in \delta^+(v)\) do
11: LABEL-UPDATE\((v, w, Q)\)

Where Algorithm 5.3 has an unspecified stop condition, Algorithm 5.6 uses a simple stop condition: both candidate sets \(Q^s\) and \(Q^f\) have to be empty. Under the assumption that the unspecified check (line 9 of Algorithm 5.3) always holds, we can draw some conclusions:

- Since we perform an exhaustive search in both the forward and backward direction, we actually implemented the Generic Shortest Path algorithm (Algorithm 3.4) in a forward and a backward manner. Thus, we are sure that, upon termination, a shortest path is found;
- Algorithm 5.6 can be implemented using either a label-setting or a label-correcting variant of the Iterate procedure (Algorithm 5.7);
- There is no additional assumption necessary on the heuristic estimators \(h^s\) and \(h^f\). It does not matter whether or not they are consistent. In fact, it even does not matter whether or not \(h^s_v > d(v, t)\) or \(h^f_v > d(s, v)\) for all or some \(v \in V\);
- Due to the exhaustive searches in both directions, we may expect that Algorithm 5.6 performs poorly, compared to a unidirectional shortest path algorithm that uses the same type of Iterate procedure.

In the remainder of this section, we will present a check for line 9 of Algorithm 5.7, which divides the total search space over the forward and the backward search such that the total running time is improved considerably. The check is based on so-called scalar projections.

Definition 5.22 The scalar projection of a vector \(a\) in the direction of vector \(b\) is defined as:

\[
\frac{a^T b}{\|b\|^2}.
\]  

(5.18)

As soon as a meeting node \(u_0\) is found, we determine in the forward search the scalar projection \(g_v^s\) of the vector \((x_{u_0} - x_v)\) for \(v \in Q^s\) in the direction of \((x_t - x_s)\) and
5.3 Bidirectional algorithms and heuristic estimators

the scalar projection $g^s_w$ of the vector $(x_{w_0} - x_w)$ for $w \in Q^t$ in the direction from $(x_s - x_t)$ in the backward search:

$$g^s_w = \frac{\langle x_s - x_t, x_{w_0} - x_w \rangle}{||x_s - x_t||_2}, \quad v \in V,$$

(5.19)

$$g^t_w = \frac{\langle x_s - x_t, x_{w_0} - x_w \rangle}{||x_s - x_t||_2}, \quad w \in V.$$

**Proposition 5.23** Let $v \in Q^t$ and $w \in Q^t$. For any node $u_0 \in V$, $g^s_v + g^t_w$ equals the scalar projection of the vector $(x_{w_0} - x_w)$ on the vector $(x_t - x_s)$.

**Proof:** This follows directly from the definition of $g^s_v$ and $g^t_w$ as given in (5.19):

$$g^s_v + g^t_w = \frac{\langle x_t - x_s, x_{w_0} - x_w \rangle}{||x_t - x_s||_2} + \frac{\langle x_s - x_t, x_{w_0} - x_w \rangle}{||x_s - x_t||_2} = \frac{\langle x_t - x_s, x_{w_0} - x_w \rangle}{||x_t - x_s||_2} + \frac{\langle x_s - x_t, x_{w_0} - x_w \rangle}{||x_s - x_t||_2} = \frac{\langle x_t - x_s, x_{w_0} - x_w \rangle}{||x_t - x_s||_2}.$$

The last term is the scalar projection of the vector $(x_{w_0} - x_w)$ on the vector $(x_t - x_s)$.

In Figure 5.7, a separating line through meeting node $u_0$ is drawn, perpendicular to the line from $s$ to $t$. For nodes $v$ and $w$, the scalar projections $g^s_v$ and $g^t_w$ are visualized. Scalar projections can be negative (see Definition 5.22). In the figure, $g^t_w$ is negative. Note that $n$ lies on the left side of the separating line, while $t$ lies on the right side.

Since the scalar projection is an orthogonal projection, the following inequality holds:

$$g^s_v + g^t_w \leq ||x_{w_0} - x_v||_2.$$

By Definition 5.11, in a road map graph, we have $c_{wv} \geq \alpha||x_{w} - x_v||_2$, for all $(v, w) \in A$ and some $\alpha \in \mathbb{R}^+$. By Proposition 5.12 we have $d(v, w) \geq \alpha||x_{w} - x_v||_2 \geq \alpha(g^s_v + g^t_w)$.

We can use $g^s_v$ and $g^t_w$ to divide the search space by introducing the following check for line 9 of Algorithm 5.7:

forward search: \quad $u_0 = \text{NIL}$ or $d^s_v + \alpha g^t_v < d^s_{u_0}$,

backward search: \quad $u_0 = \text{NIL}$ or $d^t_v + \alpha g^s_v < d^t_{u_0}$.

(5.20)

**Theorem 5.24** Let $G = (V, A)$ be a road map graph and let $h^s$ and $h^t$ be two heuristic estimators. Running Algorithm 5.6, together with the Iterate procedure as described in Algorithm 5.7 and using the check (5.20), will determine a shortest $s$-$t$ path. The meeting node $u$ is on that path.

---

3In [197] the scalar projection of $(x_{w} - x_{w_0})$ on $(x_s - x_{w_0})$ is used, together with the scalar projection of $(x_{w} - x_{w_0})$ on $(x_t - x_{w_0})$. If $x_{w_0}$ is not lying on the line from $x_s$ to $x_t$, this may lead to incorrect shortest paths. In a private communication, Whangbo agreed that our projections should be used.
Proof: Since $G$ is strongly connected and after initializations $Q^s \neq \emptyset$ and $Q^t \neq \emptyset$, we are sure that at least one meeting node will be found eventually. By Algorithm 5.7, the first meeting node found is stored as $u_0$. From that moment on, the check (5.20) might fail. If the check fails for a node $v$ with a non-sharp label (see Definition 3.28), it does not matter at all since the current label value of node $v$ will be improved in a later iteration. Suppose node $v$, having a sharp label, is selected as a pivot node during a search iteration in one direction and the check fails. We will prove that, if a shortest $s$-$t$ path containing node $v$ exists, that particular path will be found during search iterations in the opposite direction, or that another shortest path (not necessarily via node $v$) will be found.

For ease of notation we assume that during a forward search iteration, a pivot node $v$ with a sharp label is selected for which $d^s_v + \alpha g^s_v \geq d^u_0$. If the check fails during a backward search iteration for a pivot node with a sharp label, we can use a similar argumentation. Suppose there is a shortest $s$-$t$ path $P$, containing node $v$. Thus, $P = (s = v_0, (v_0, v_1), v_1, \ldots, v_i = v, (v_i, v_{i+1}), v_{i+1}, \ldots, v_{k-1}, (v_{k-1}, v_k), v_k = t)$. Assume that node $w \in \{v_i, v_{i+1}, \ldots, v_k\}$ is selected as pivot node during a backward iteration. If $d^w_w$ is non-sharp ($d^w_w > d(w, t)$), then it does not matter whether or not the check fails since, in a later backward iteration, the label $d^w_w$ will be updated. We will prove that the check in the backward iteration always holds if $d^w_w$ is a sharp label. We have:

$$d^w_v + \alpha(g^s_v + g^t_w) + d^w_w = d(s, v) + \alpha(g^s_v + g^t_w) + d(w, t) \leq d(s, v) + d(v, w) + d(w, t) \leq d^u_s + d^t_w.$$
The first equality holds due to the sharp labels \( d_s \) and \( d_t \). The first inequality holds by the corollary of Proposition 5.23. Since \( d(s, v) + d(v, w) + d(w, t) \) equals the length of path \( P \), and \( P \) is a shortest \( s \)-\( t \) path, the last inequality holds. Since the check failed in the forward iteration, having node \( v \) as pivot node, we have:

\[
d_{u_0}^t + d_w^t + \alpha g_{w}^t \leq d_{v}^t + \alpha g_{w}^u + d_{u}^w \leq d_{u}^t \leq d_{u_0}^t + d_{u_0}^t.
\]

Thus, \( d_{w}^t + \alpha g_{w}^t \leq d_{u_0}^t \). If \( d_{w}^t + \alpha g_{w}^t < d_{u_0}^t \) then the check holds in the backward iteration. If \( d_{w}^t + \alpha g_{w}^t = d_{u_0}^t \) we have:

\[
d_{u_0}^t + d_{u_0}^t = d_{u_0}^t + d_{u}^w + \alpha g_{w}^t \leq d_{v}^t + \alpha g_{w}^u + d_{u}^w + \alpha g_{w}^t \leq d_{u}^t \leq d_{u_0}^t + d_{u_0}^t.
\]

Hence, the length of \( P \) is equal to the shortest \( s \)-\( t \) path containing node \( u \), which we already found.

Dividing the search space based on scalar projections as described above, usually reduces the search space considerably. This behavior motivates the name of this subsection: divide and conquer.

In Figure 5.8, the progress in research on symmetric approaches is visualized. It starts with Pohl’s stop condition in part (a). Next, part (b) shows the effect of nipping and pruning as introduced by Kwa (see Section 5.3.1 for more details). Part (c) is based on the new lower bound of Pijls and Post (Section 5.3.3). Finally, in part (d), the search space based on scalar projections is visualized.

The search space of part (d) is clearly smaller than the search space of Figure 5.6, which was based on balanced heuristic estimators. Thus, we conclude that it is likely that the use of scalar projections and symmetric heuristic estimators is preferred over the use of balanced heuristic estimators. As said before, the use of balanced heuristic estimators is preferred in most of the related literature until now.
Figure 5.8: Progress in search space reduction using a symmetric approach.
Preprocessing the graph

In this chapter, we will review several preprocessing techniques that can be used to improve the calculation time for any SPP-algorithm. The idea of these preprocessing techniques is to add some extra data to the original graph, which can be used during the execution of an SPP-algorithm.

Preprocessing techniques can be characterized by the amount of extra data included in the graph, the calculation time to obtain this extra data, and of course the average speed-up that can be achieved by using this extra data. Furthermore, the sensitivity of the calculated extra data to (minor) changes in the graph should be taken into account. We distinguish two main types of sensitivity: Preprocessing techniques that are insensitive to changes in the arc costs $c_{vw}$, and those that are sensitive to changes in the arc costs.

6.1 Test environment

To illustrate the effectiveness of several preprocessing techniques, we consider a road map provided by TomTom. In this section, we briefly describe this graph. We also provide the necessary specifications and type of computer and compiler used to perform our tests.

6.1.1 Road map of TomTom

We consider the combined road map of Germany, the Netherlands, Belgium and Luxembourg, provided by TomTom (Multinet, July 2010). This is an undirected graph $G^{TA} = (V^{TA}, E^{TA})$. To each edge $e \in E^{TA}$, several attributes are assigned. These attributes provide detailed information about the road described by edge $e$ like road condition, road name, road length et cetera.
The graph is undirected, therefore an edge \( e \) adjacent to the nodes \( v \) and \( w \) can be referenced to as either \((v, w)\) or \((w, v)\). The ‘head’ attribute \( h_e \) of such an edge \( e \) defines a positive direction for the undirected edge \( e \):

\[
h_e = \begin{cases} 
  v & \text{the positive direction on edge } e \text{ is from } w \text{ to } v, \\
  w & \text{the positive direction on edge } e \text{ is from } v \text{ to } w.
\end{cases}
\]

Since there are two possible directions for traversing edge \( e \), the opposite of the positive direction is called the negative direction. An important attribute is the ‘direction of flow’ attribute \( f_t \). Here, \( t \) is a certain vehicle type (car, truck, bus, bicycle, etc.):

\[
f_t^e = \begin{cases} 
  0 & \text{edge } e \text{ is prohibited for vehicle type } t, \\
  1 & \text{edge } e \text{ can be traversed by vehicle type } t \text{ in the positive direction,} \\
  2 & \text{edge } e \text{ can be traversed by vehicle type } t \text{ in the negative direction,} \\
  3 & \text{edge } e \text{ can be traversed by vehicle type } t \text{ in both directions.}
\end{cases}
\]

For each vehicle type \( t \), a directed graph \( G_t = (V_t, A_t) \) can be constructed as follows: \((v, w) \in A_t\) if and only if \( e = (v, w) \in E^{TA}\) and either \( h_e = w \) and \( f_t^e \in \{1, 3\} \) or \( h_e = v \) and \( f_t^e \in \{2, 3\} \). Node \( v \in V^{TA} \) belongs to node set \( V_t \subseteq V^{TA} \) if there exists either an arc \((v, w) \in A_t\) or an arc \((w, v) \in A_t\) for any \( w \in V^{TA}\).

The graph \( G_t \) might not be connected. We extract the largest component \( G_t = (V_t, A_t) \) from the graph \( G_t \). This extraction can easily be done using the STRONG-CONNECT procedure, described by Tarjan [184], in \( O(|V_t| + |A_t|) \) time. In our environment (see next section), this takes only 8.5 seconds for vehicle type ‘car’.

In this thesis, we are especially interested in the vehicle type ‘car’. For this vehicle type, several statistics are mentioned in Table 6.1 with regards to the relevant graphs.

### Table 6.1: Graph statistics for vehicle type car

| Graph \( G^{TA} \) | \( |V^{TA}| \) | 9,378,976 | Graph \( G_{\text{car}} \) | \( |V_{\text{car}}| \) | 8,682,332 | Graph \( G_{\text{car}} \) | \( |V_{\text{car}}| \) | 8,678,011 |
|-------------------|-------------|-----------|------------------|--------------|-----------|------------------|--------------|-----------|
| \( |E^{TA}| \)     | 11,389,880  |           | \( |A_{\text{car}}| \) | 19,581,272  |           | \( |A_{\text{car}}| \) | 19,572,341  |

#### 6.1.2 Hardware and compiler

Several of the preprocessing techniques described in this chapter are tested on the previously mentioned graph. Running times are sometimes presented. These running times are based on a Dell Precision 490 with a 2.66 Ghz processor and 3.25 Gb memory. All algorithms are coded in Delphi 2010.
6.2 Static graph preprocessing

Preprocessing techniques that are insensitive to changes in the arc costs \( c_{vw} \) perform a ‘static graph preprocessing’, which can be used in any SPP-algorithm for any set of arc costs. In [121], two techniques are presented. These techniques are based on a connected undirected graph.

To illustrate the techniques, we use the road map of TomTom (see Section 6.1.1). The undirected graph \( G = (V, E) \) we consider is the undirected version of \( G_{\text{car}} \). Thus, \( V = \overline{V}_{\text{car}} \) and \((v, w) \in E\) if and only if either \((v, w) \in A_{\text{car}}\) or \((w, v) \in A_{\text{car}}\).

6.2.1 Quarters

In a graph obtained from a road map, the situation often occurs that some nodes are only reachable from \( s \) when the path contains a specific edge. Consider for example a dead-end road. Sometimes even a whole quarter can only be reached by a specific road. The corresponding edges in the graph are called bridges.

**Definition 6.1** Let \( G = (V, E) \) be an undirected graph. A bridge is an edge \((v, w) \in E\) where \((v, w) \in P\) for every path \( P \) from \( v \) to \( w \).

The bridges of a graph can easily be found by using a depth-first search algorithm. In fact, any graph search algorithm can be used in an efficient algorithm, presented by Tarjan [185].

Bridges are closely related to articulation nodes:

**Definition 6.2** Let \( G = (V, E) \) be an undirected graph. If there exists a set of three distinct nodes \( s, t, v \in V \), such that \( v \) is on any \( P_{st} \) path, and at least one such path exists, then \( v \) is called an articulation node.

In related literature, another definition for a bridge is widely used: A bridge is an edge, deletion of which results in a graph with more components than the original graph. For an articulation node, a similar definition exists: An articulation node is a node that, when deleted along with its incident edges, results in a graph with more components than the original graph. Although these definitions are equivalent to Definition 6.1 and Definition 6.2, we find the latter more clarifying for our purpose.

**Definition 6.3** Let \( G = (V, E) \) be an undirected graph. \( G \) is biconnected if it has no articulation node.

**Definition 6.4** Let \( G = (V, E) \) be an undirected graph. A biconnected component of \( G \) is a maximal set of edges \( E' \subset E \), such that any two edges in the set \( E' \) lie on a common simple cycle.
In Figure 6.1 [35], these definitions are illustrated. Tarjan [184] presented the Bt-
connect procedure, which determines all biconnected parts of an undirected graph
\( G = (V, E) \) in \( O(|V| + |E|) \) time.

![Figure 6.1](image)

**Figure 6.1:** The bridges, articulation nodes and biconnected components
of a connected, undirected graph. The articulation nodes are
black, the bridges are the heavily shaded edges and the biconnected
components are the edges in the numbered regions.

**Definition 6.5** The nodes connected to the largest biconnected component of the
graph \( G = (V, E) \) (with respect to the number of connected nodes to the biconnected
component) are called core nodes. The set of core nodes is denoted by \( V^0 \subset V \). A
node \( v \in V \setminus V^0 \) is called a quarter node.

After determining all biconnected parts of an undirected graph, it is easy to deter-
mine the set of core nodes \( V^0 \).

In Figure 6.2, a part of the Dutch city Hilversum is shown. Quarter nodes are
highlighted with blue dots. The red dots depict unreachable nodes (by car), thus
nodes in \( V^{TA} \setminus V_{car} \).

### 6.2.2 Bypass degree 2 nodes

One of the easiest preprocessing steps is based on the observation that there are
many nodes with degree 2 in GIS-oriented data. These nodes often mark a change
of attributes for the adjacent edges. (Note that the graph is considered undirected.)
The shortest path may be found sooner if we reduce the number of nodes and arcs
that have to be searched. This can be done by adding shortcuts to the graph. A
shortcut is a simple path for which every node in that path (except for the first and
the last node) has degree 2. The intermediate nodes on the path are bypassed by
the shortcut. See Figure 6.3 for an illustration.

Degree 2 nodes can easily be bypassed through the procedure described in Algo-
rithm 6.1. This algorithm takes a biconnected undirected graph \( G = (V, E) \) as input.
Normally, we first extract the largest biconnected component \( G^0 = (V^0, E^0) \) from a
graph \( G = (V, E) \) (see Section 6.2.1) and \( G^0 \) is used as input for Algorithm 6.1.
6.2 Static graph preprocessing

Figure 6.2: Quarter nodes (blue) in a part of Hilversum

Figure 6.3: Bypassing degree 2 nodes ($v_1$ and $v_2$) with a shortcut (the bold edge)
Algorithm 6.1: BypassDegree2($\mathcal{G} = (\mathcal{V}, \mathcal{E})$)

Require: $\mathcal{G}$ is biconnected and contains a node with degree $\neq 2$

1: $S \leftarrow \emptyset$
2: foreach $v \in \mathcal{V}$ do
3:   if $v$ has degree 2 then
4:     $p \leftarrow v$ and $S \leftarrow (u, \{u, v\}, v)$
5:     while $u$ has degree 2 do
6:       if $u' = p$ then
7:         $p \leftarrow u$ and $u \leftarrow u''$
8:       else
9:         $S \leftarrow (u, \{u, p\}, p) + S$
10:        $p \leftarrow v$ and $S \leftarrow S + (v, \{v, w\}, w)$
11: while $w$ has degree 2 do
12:   if $w' = p$ then
13:     $p \leftarrow w$ and $w \leftarrow w''$
14:   else
15:     $S \leftarrow S + (p, \{p, w\}, w)$
16:     $S \leftarrow S \cup \{S\}$
17: Create a new edge $\{u, w\}$ and add this to $\mathcal{E}$
18: foreach $\{v, w\} \in S$ do
19:   $\mathcal{E} \leftarrow \mathcal{E} - \{v, w\}$
20: foreach intermediate node $v \in S$ do
21:   $\mathcal{V} \leftarrow \mathcal{V} - \{v\}$
22: return $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ and $S$
Let us briefly describe Algorithm 6.1. The main “for-loop” in this algorithm (line 2) determines for each node \( v \in V \) whether or not it is a degree 2 node (line 3). If \( v \) has degree 2, it is connected to two other nodes \( u \) and \( w \). These nodes, again, might have degree 2. This way, a search is started in both the direction of node \( u \) (lines 5–12) as well as in the direction of node \( w \) (lines 13–20) until a node is found without degree 2. Between those two non-degree 2 nodes, a shortcut \( S \) is constructed (during the searches). This shortcut is added to the collection of shortcuts \( S \) (line 21) and the graph is updated (lines 22–26). There are some more detailed remarks we should mention:

- Since \( G \) is biconnected and has at least one node with degree \( \neq 2 \), no cycle containing only degree 2 nodes exists. Accordingly, both “while-loops” are finite. Furthermore, we know \( u \neq w \) in line 22.
- As soon as a shortcut is created, a new edge \( \{u, w\} \) is added to \( E \). But \( E \) might already contain an edge \( \{u, w\} \), so we may end up with multiple edges at the end of the algorithm. Since we are unaware of the actual arc costs that about to be used, we can not decide which edge should be kept. Therefore, we are forced to allow multiple edges at this moment. As soon as a shortest path algorithm is started, we can easily get rid of multiple arcs by ignoring those that are not the cheapest.

At the end of the algorithm, the graph \( G' = (V', E') \) is returned in combination with a collection of shortcuts \( S \). Here \( V' \subset V \) and each \( v \in V' \) has degree \( \neq 2 \). Furthermore, all paths in \( G \) between any pair of nodes \( v, w \in V' \) are preserved in \( G' \). In Figure 6.4, the green nodes are the bypassed degree 2 nodes in a section of the largest biconnected component of our input graph (see Section 6.1.1).

### 6.2.3 Partitioning the node set \( V \)

As soon as we determined a set of core nodes \( V^0 \), by determining quarters (Section 6.2.1) and bypassing degree 2 nodes (Section 6.2.2), we can easily create a partition \( V^1, V^2, \ldots, V^K \) of \( V \setminus V^0 \), where \( v, w \in V^k \), \( 1 \leq k \leq K \) if and only if there exists a path from \( v \) to \( w \) that contains only nodes in \( V \setminus V^0 \). Such a partition can be constructed using Algorithm 6.2.

Suppose we want to determine the shortest path from node \( s \in V^k \) to node \( t \in V^j \). For any path \( P_{st} \), the following condition holds. If \( v \in P_{st} \), then \( v \in V^0 \cup V^k \cup V^j \). Instead of using the full graph \( G = (V, \mathcal{A}) \) we can use the subgraph \( \overline{G} = (\overline{V}, \overline{\mathcal{A}}) \), where \( \overline{V} = V^0 \cup V^k \cup V^j \) and \( \overline{\mathcal{A}} = \{(v, w) \in \mathcal{A} | v, w \in \overline{V}\} \), to determine the shortest path from node \( s \) to node \( t \).

### 6.2.4 Summary

To summarize the preprocessing methods in this section, we outline all the steps in Algorithm 6.3. Table 6.2 shows several statistics for each step in Algorithm 6.3.
Algorithm 6.2: PARTITIONING($\mathcal{G}, V^0$)

**Require**: $\mathcal{G} = (\mathcal{V}, \mathcal{A}), V^0 \subset \mathcal{V}$

1. $k \leftarrow 0$
2. foreach $v \in \mathcal{V}$ do
3.    if $v \in V^0$ then
4.        mark $v$ as assigned
5.    else
6.        mark $v$ as unassigned
7.    foreach $v \in \mathcal{V}$ do
8.       if $v$ is unassigned then
9.          $k \leftarrow k + 1$
10.         $V^k \leftarrow \emptyset$
11.        DFS($v$)

**procedure**: DFS($v$)

12. $V^k \leftarrow V^k \cup \{v\}$
13. mark $v$ as assigned
14. foreach ($v, w) \in \mathcal{A}$ do
15.    if node $w$ is unassigned then
16.       DFS($w$)
Based on these statistics, we conclude that for any source node \( s \in \mathcal{V} \) and any destination node \( t \in \mathcal{V} \), the sub graph needed to solve the SSSD-SPP (see Section 6.2.3) has no more than \( 2,657,508 + 2 \times 2,127 \) nodes.

**Algorithm 6.3: Static Preprocessing**\((\mathcal{G} = (\mathcal{V}, \mathcal{E}))\)

1. Call Biconnect\((\mathcal{G})[184]\)
2. Extract largest biconnected component \( \mathcal{G}^0 = (\mathcal{V}^0, \mathcal{E}^0) \)
3. Call BypassDegree2\((\mathcal{G}^0)\)
4. Call Partitioning\((\mathcal{G}, \mathcal{V}^0)\) (Algorithm 6.2)

<table>
<thead>
<tr>
<th>Action</th>
<th>Entity</th>
<th>Amount</th>
<th>%</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>(</td>
<td>\mathcal{V}_{\text{car}}</td>
<td>)</td>
<td>8,678,011</td>
</tr>
<tr>
<td>Biconnect</td>
<td>Quarter nodes</td>
<td>2,503,779</td>
<td>29</td>
<td>11.8</td>
</tr>
<tr>
<td>BypassDegree2</td>
<td>Degree 2 (non quarter) nodes</td>
<td>3,516,724</td>
<td>40</td>
<td>4.1</td>
</tr>
<tr>
<td></td>
<td>Core nodes ( \mathcal{V}^0 )</td>
<td>2,657,508</td>
<td>31</td>
<td></td>
</tr>
<tr>
<td>Partitioning</td>
<td>Number of subsets ( \max_{1 \leq k \leq K}</td>
<td>\mathcal{V}^k</td>
<td>)</td>
<td>1,672,076</td>
</tr>
<tr>
<td></td>
<td>(</td>
<td>\mathcal{A}_{\text{car}}</td>
<td>)</td>
<td>1,363,747</td>
</tr>
<tr>
<td></td>
<td>Arcs + shortcut-arcs</td>
<td>22,299,835</td>
<td>114</td>
<td></td>
</tr>
</tbody>
</table>

We conclude that we can reduce the number of nodes needed for a SSSD-SPP to 31%, at the cost of an easily implemented preprocessing algorithm that runs in less than 20 seconds. The extra amount of data needed is an integer \( v^k \) for each node \( v \in \mathcal{V} \), denoting the subset \( \mathcal{V}^k \) of the partition of \( \mathcal{V} \) where \( v \) is included. Furthermore, 14% more arcs are included in the graph.

### 6.3 Dynamic graph preprocessing

Many of the preprocessing techniques described below need the notion of **canonical path**.

**Definition 6.6** [90] A canonical path is a shortest path with additional properties:

1. A canonical path is a simple shortest path.
2. For every pair \( s, t \), there is a unique canonical path between \( s \) and \( t \).
3. A subpath of a canonical path is a canonical path.
4. It is possible to implement any SPP algorithm in such a way that it always finds canonical paths.
Since we consider only graphs without negative and zero cycles, all shortest paths are simple paths.

Furthermore, we need the notion of a metric on the node set $\mathcal{V}$.

**Definition 6.7** A metric (or distance function) $m$ on the node set $\mathcal{V}$ is a function $m : \mathcal{V} \times \mathcal{V} \rightarrow \mathbb{R}$. For all $u, v, w \in \mathcal{V}$, this function is required to satisfy the following conditions:

1. $m(v, w) \geq 0$.
2. $m(v, w) = 0 \iff v = w$.
3. $m(v, w) = m(w, v)$.
4. $m(v, w) \leq m(v, u) + m(u, w)$.

The first condition in Definition 6.7 is implied by the others. However, please note that the cost function $C$ that assigns a cost $c_{vw}$ to any arc $(v, w) \in \mathcal{A}$, is not necessarily a metric. Especially condition three (symmetry) is not always obeyed.

### 6.3.1 Graph contraction

#### 6.3.1.1 Node reduction

Bauer and Delling [12] describe a preprocessing technique that is closely related to the ideas of Sanders and Schultes [171]. The graph is contracted by iteratively bypassing nodes until no node is bypassable anymore.

**Definition 6.8** Bypassing node $v$ in $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ means the creation of the contracted graph $\overline{\mathcal{G}} = (\overline{\mathcal{V}}, \overline{\mathcal{A}})$, with $\overline{\mathcal{V}} = \mathcal{V} \setminus \{v\}$ and $\overline{\mathcal{A}} = \{(v, w) \in \mathcal{A} | v, w \in \overline{\mathcal{V}}\} + \mathcal{S}_v$.

Here, $\mathcal{S}_v$ is a set of shortcuts $(u, w)$ with $(u, v) \in \delta^-(v)$ and $(v, w) \in \delta^+(v)$ and $u \neq w$ with respect to the original graph $\mathcal{G}$. If for any $(u, v) \in \delta^-(v)$ and for any $(v, w) \in \delta^+(v)$ arc $(u, w)$ is included in $\mathcal{A}$, no shortcut is created between $u$ and $w$. Thus, $(u, w) \notin \mathcal{S}_v$ but the cost $c_{uw}$ of arc $(u, w) \in \overline{\mathcal{A}}$ is updated to $c_{uw} \leftarrow \min(c_{uw}, c_{uv} + c_{vw})$.

Figure 6.5 shows an example of bypassing. Here, $\mathcal{S}_v = \{(u, w), (u, y), (x, u), (x, w)\}$. Furthermore, the arc cost $c_{xy}$ is decreased to 2.

Each shortcut represents a path in the original graph $\mathcal{G}$. The number of arcs of the path a shortcut represents is called the **hop number** of the shortcut.

A node $v \in \mathcal{V}$ is said to be bypassable if $|\mathcal{S}_v| \leq c \times (|\delta^-(v)| + |\delta^+(v)|)$, where $c$ is a tunable **contraction parameter** and $\mathcal{S}_v$ does not contain a shortcut with a hop number greater than a certain $h$. The idea to bound the hop number of a shortcut was proposed by Delling et al. [52].

As soon as node $v$ is bypassed, the degree of the neighbor nodes (nodes connected to node $v$ in its underlying undirected graph) might be changed, and so will their
6.3 Dynamic graph preprocessing

bypassability. Therefore, the order in which nodes are bypassed changes the resulting contracted graph. Bauer and Delling use a heap to determine the next bypassable node. The key of a node $v$ within the heap is $H \times |S_v|/(|\delta^-(v)| + |\delta^+(v)|)$, where $H$ is the hop number of the hop-maximal shortcut that would be added if $v$ were bypassed. Smaller keys have higher priority. The idea to control the order in which the nodes are bypassed using a heap is found in [91].

If a node $v$ is not bypassable, node $v$ is a core node. The set of core nodes is called $V^0$.

As described in Section 6.2.3, we can create a partition of $V$ and use this partition to decrease the number of iterations needed to calculate the shortest path from a node $s$ to node $t$.

6.3.1.2 Arc reduction

Bauer [13] introduces a second step to be taken after the node reduction step. The idea is to decrease the number of arcs in the core graph $G^0 = (V^0, A^0)$, where $V^0$ is the set of core nodes and $A^0 = \{ (v, w) \in A | v, w \in V^0 \} + S$, where $S$ is the set of all shortcuts added between two core nodes during the node reduction step.

The arc reduction algorithm described here is similar to the work of Schultes and Sanders [176]. For each $v \in V^0$, a (canonical) shortest path tree is grown, rooted at $v$. The growth is stopped as soon as all neighbors (nodes connected to $v$) have been settled. Then, for all neighbors $w$ of node $v$, it is checked whether or not $v$ is the parent of $w$ in the grown partial shortest path tree. If $v$ is not the parent, $(v, w)$ is removed from $A^0$, because the shortest path from $v$ to $w$ does not include $(v, w)$.

6.3.1.3 Example

The graph contraction, as described in this section, can be tuned with the contraction parameter $c$ and the maximal hop size $h$. In Table 6.3, the statistics are shown for the same pairs of these parameters, as reported in [13]. The statistics are based on our input graph, consisting of the road map of Germany, the Netherlands, Belgium and Luxembourg. The CPU time is in seconds.
As shown in Table 6.3, the graph is contracted even with the parameters $c = 0$ and $h = 0$. Since only new shortcuts are added, it might happen that $S_v = \emptyset$, which means that node $v$ is bypassable. The first row in the table shows the situation where the node reduction step is skipped and only the arc reduction is executed. The number of arcs $|A|$ after the arc reduction step is also presented as a percentage of the number of arcs of the original input graph (19,742,501).

In [13], the running times for 10,000 SSSD problems are compared for each pair of parameters, shown in Table 6.3. It is observed that setting $c = 2.5$ and $h = 50$, leads to the best results. For these parameters, the core nodes are shown in the region of Amersfoort in Figure 6.6.

| $c$ | $h$ | $|\mathcal{V}_0|$ | $\max |\mathcal{V}_k|$ | $|A|$ | CPU | Arc reduction | % | CPU |
|-----|-----|-----------------|-----------------|-------|-----|-------------|---|-----|
| -   | -   | 8,678,011       | 100.0           | 0     | 19,742,501 | 0 | 99.1 | 41 |
| 0.0 | 0   | 6,360,689       | 73.3            | 103   | 19,704,032 | 35| 99.1 | 29 |
| 0.5 | 10  | 2,361,213       | 27.2            | 202   | 22,383,317 | 89| 111.4| 30 |
| 1.0 | 20  | 476,487         | 5.5             | 828   | 22,352,460 | 220| 109.4| 22 |
| 2.0 | 30  | 251,246         | 2.9             | 15,216| 21,966,395 | 378| 106.1| 27 |
| 2.5 | 50  | 153,359         | 1.8             | 26,369| 21,593,444 | 549| 104.8| 35 |
| 3.0 | 75  | 104,408         | 1.2             | 45,272| 21,420,792 | 803| 103.7| 39 |
| 5.0 | 100 | 75,868          | 0.9             | 89,358| 21,340,996 | 1,327| 103.1| 50 |

**Table 6.3:** Graph statistics for varying contraction parameters $c$ and $h$ (CPU times in seconds)

![Figure 6.6: Core nodes in the region of Amersfoort](image)
6.3 Dynamic graph preprocessing

6.3.2 Reaches

6.3.2.1 Arc radii

Ertl [65] introduced the concept of arc radii. For each arc $(v, w) \in A$, an associated radius $r_{vw}$ is calculated as follows:

$$r_{vw} \leftarrow \max \{ \min \{ m(s, v), m(v, t) \} \mid (v, w) \in P_{st}, s, t \in V \}.$$  \hspace{1cm} (6.1)

Here, $P_{st}$ is the canonical path between $s$ and $t$ and $m$ is a metric (see Definition 6.7). It should be noted that this metric does not necessarily have a relation to the arc costs $c_{vw}$. For example, $m(v, w)$ can be the Euclidean distance between the nodes $v$ and $w$ if the graph is projected on a planar map, while $c_{vw}$ is the time needed to travel along edge $(v, w)$.

When looking for a shortest path from source node $s$ to destination node $t$, an arc $(v, w)$ is only considered as a possible arc on the path if either $s$ or $t$ lies inside this radius of the arc, thus $\min (m(s, v), m(v, t)) \leq r_{vw}$. If both distances are greater than the arc radius, the arc is simply ignored. To decrease the running time a bit more, the outgoing arcs of a node are sorted by descending radius.

To calculate the radii of all arcs $(v, w) \in A$, we need to determine the (canonical) shortest path tree rooted at $s$ for all $s \in V$. This means that the calculation of all radii is a very time-consuming process. Ertl presents a heuristic algorithm to calculate good upper bounds for each $r_{vw}$ by dividing the graph in several overlapping rectangles. The preprocessing time he reported is however still 22 hours on a graph of Germany with $|A| = 241,516$.

6.3.2.2 Node reaches

Gutman [99] introduced a very similar concept called node reaches. Before we give a formal definition of the reach of a node, we need to introduce an additional notation. For a (simple) path $P$ and a metric $m$, we use $m(P)$ to denote the sum of $m(v, w)$, where $(v, w) \in P$. If $P$ contains only one node, $m(P) = 0$.

For each node $v \in V$, an associated reach $r_v$ is defined as follows:

$$r_v \leftarrow \max \{ \min \{ m(P_{sv}), m(P_{vt}) \} \mid v \in P_{st}, s, t \in V \}.\hspace{1cm} (6.2)$$

Here, $P_{st}$ is the canonical shortest path between node $s$ and $t$. Since node $v \in P_{st}$, we can divide the path $P_{st}$ in two (canonical) subpaths $P_{sv}$ and $P_{vt}$.

We assume there is a consistent heuristic estimator $h$ for the metric $m$, so $h(v) - h(w) \leq m(v, w)$ for any two nodes $v, w \in V$. A node $v$ is only considered as a possible node on the shortest path from source node $s$ to destination node $t$, if either $m(P_{sv}) \leq r_v$ or $h(v) - h(t) \leq r_v$. 
To calculate the node reaches of all nodes $v \in V$, we need to determine the (canonical) shortest path tree rooted as $s$ for all $s \in V$. Accordingly, the time needed to calculate all reaches is comparable to the time needed to calculate the arc radii in Ertl’s algorithm. Like Ertl, Gutman presented a heuristic algorithm to calculate good upper bounds for each $r_v$. His heuristic algorithm is a kind of bootstrapping\(^1\) algorithm. First, small reach bounds are calculated for some nodes. This information is subsequently used to compute larger reach bounds for certain other nodes. This bootstrapping from low to high reach bounds is repeated until reach bounds for most nodes have been computed. To the remaining nodes, infinite reach bounds are assigned. Gutman reported a preprocessing time of 161 minutes to calculate the reach bounds for a graph of San Francisco Bay Area with $|V| = 393,368$.

Since the metric $m$ is not related to the arc costs $c_{vw}$, additional data should be included in the graph. To determine for example the Euclidean between two nodes, we need to know the coordinates of each node.

Goldberg et al. [90] improved the ideas of Gutman. First, they used node reaches that are not based on a metric $m$ but on the actual shortest path lengths:

$$r'_v \leftarrow \max \left\{ \min \{ \ell(P_{sv}), \ell(P_{vt}) \} \mid v \in P_{st}, s, t \in V \right\}.$$  

(6.3)

Let $h$ be an optimistic heuristic estimator (See Definition 5.3). The outgoing arcs of node $v$ are not scanned if $d_v > r'_v$ and $h_v > r'_v$.

Furthermore, they incorporated this idea of pruning nodes by using node reaches in a bidirectional label-setting algorithm. They observed that in a bidirectional label-setting algorithm, no heuristic estimator is needed to profit from node reaches. Suppose node $v$ is going to be scanned in the primary direction. Node $v$ can be pruned, if the following conditions hold:

$$r'_v < d_v, \quad r'_v < \min \{ d_w \mid w \in \bar{Q} \}.$$  

(6.4)

In fact, they observed that for any node $v \notin \bar{R}$, $\min(d_w \mid w \in \bar{Q}) \leq d(v, t)$. This is a special case of our Proposition 5.20, with $h^*_v = h^*_t = 0$ for all $v \in V$.

Instead of pruning node $v$ when both conditions of (6.4) hold, they proposed a so-called self-bounding algorithm where a node is pruned if $r'_v < d_v$. If $d(v, t) = r'_v$, node $v$ will eventually be scanned in the opposite direction if the following stop condition is used: the search in a given direction is stopped when either $Q = \emptyset$ or $\min(d_w \mid w \in Q) \geq (d_u + d_{tu})/2$, where $u$ is the best meeting node found at that time.

Finally, they extended the concept of node reaches to arc reaches. Arc reaches are considered to be more powerful since the reach of an arc $(v, w)$ may be much smaller than $r'_v$ and $r'_w$. The arc reaches in [90] areconceptually the same as the arc radii in [65]. As far as we know, this relation is not reported in the literature.

\(^1\)Bootstrapping refers to a self-sustaining technique that proceeds without external help.
Figure 6.7: Pruning nodes based on node reaches.
To demonstrate the power of node reaches, we visualize the searched arcs in Figure 6.7. Part (a) is based on unidirectional A* search. Part (b) is based on bidirectional Dijkstra (no heuristic estimators). Part (c) shows the searched arcs for bidirectional A* with balanced estimators. Part (d) is based on bidirectional A* search with symmetric estimators and scalar projections. All estimators use Euclidean estimates (see Definition 5.7). Instead of visualizing the searched area, we decided to draw all searched arcs to emphasize the large gaps in the area investigated.

In [90], the calculation of upper bounds on node reaches is optimized by adding shortcuts to the graph. Calculating exact node reaches means that, for any node \( s \in \mathcal{V} \), the canonical shortest path tree has to be determined. In [91], it is argued that re-using canonical shortest path subtrees found earlier might lead to faster calculation of the node reaches. More experimental results are presented in [92].

### 6.3.3 Landmarks

In 2004, Goldberg and Harrelson [89] introduced the concept of landmarks. A small set of nodes \( L \subset \mathcal{V} \) is selected. For each node \( v \in \mathcal{V} \) and each landmark \( L \in \mathcal{L} \), the length \( h^+_L(v) \) of a shortest path from node \( v \) to landmark \( L \) is calculated during preprocessing, as well as the length \( h^-_L(v) \) of a shortest path from landmark \( L \) to node \( v \). By the triangle inequality, we have \( h^+_L(v) - h^-_L(w) \leq d(v, w) \) and \( h^-_L(w) - h^+_L(v) \leq d(v, w) \) for all \( v, w \in \mathcal{V} \), where \( d(v, w) \) denotes the length of a shortest path from \( v \) to \( w \) (as shown in Figure 6.8).

![Figure 6.8: Landmarks and their triangle inequalities.](image)

From landmark \( L \), we obtain the lower bound \( h_L(v, t) \) for \( d(v, t) \):

\[
h_L(v, t) = \max \{ h^+_L(v) - h^-_L(t), h^-_L(t) - h^+_L(v) \}. \tag{6.5}
\]

To get the tightest lower bound \( h(v, t) \) from \( L \), one can take the maximum of \( h_L(v, t) \) for all \( L \in \mathcal{L} \).

\[
h(v, t) = \max_{L \in \mathcal{L}} h_L(v, t). \tag{6.6}
\]

For any \( L \in \mathcal{L} \), the landmark estimator \( h_L(v, t) \) is consistent. Thus, by Theorem 5.8, the estimator (6.6) is consistent. We may incorporate the landmark-estimator in the \( A^* \) algorithm (Algorithm 5.1). The resulting algorithm is called ALT (the name is an abbreviation of \( A^* \) search, Landmarks and the Triangle inequality).
Each landmark needs a considerable amount of memory to store the distances from and to any node in $\mathcal{V}$. Therefore, the size $|\mathcal{L}|$ is commonly small. Given a maximum number of landmarks that can be chosen, the question is how these landmarks should be chosen in such a way that they are most effective during the shortest path calculations. In [89] and [96], several strategies are discussed that can be used to select the nodes to be included in the set of landmarks $\mathcal{L}$. We briefly mention these strategies:

**Random** The landmarks are chosen randomly from $\mathcal{V}$.

**Farthest** Pick the first landmark $L$ randomly from $\mathcal{V}$ and set $\mathcal{L} \leftarrow \{L\}$. Proceed in iterations until $|\mathcal{L}|$ equals the number of landmarks we are looking for. At each iteration, we add to $\mathcal{L}$ a landmark $v$ that lies farthest from all landmarks already chosen:

$$v = \arg \max_{v \in \mathcal{V}} \min \{d(v, L) | L \in \mathcal{L}\}$$

In [96], the original version of the farthest landmark selection strategy is extended to a version where the distance $d(v, L)$ to a landmark is not measured in the length (cost) of a path, but in the number of arcs on the path. Thus, instead of using the original $c_{vw}$ for each arc, we take $\tilde{c}_{vw} = 1$ for all arcs. Now we can use a breadth first search algorithm to determine the value of $\tilde{d}(v, L)$, see Section 3.1.4. Landmarks are picked from dense regions in the graph this way.

**Planar** Suppose the coordinates of each node are known. The first landmark is chosen near the center of the graph. The graph is divided into sectors originating from this landmark. The farthest node in each sector is selected as a landmark. If the new landmark lies close to the sector boundary, adjacent nodes of the neighboring sector are skipped.

In [96], it is stated that slightly better results are obtained if one chooses the modified arc costs $\tilde{c}_{vw} = 1$ for all arcs, to determine the farthest nodes in each sector. Furthermore, instead of choosing the first landmark near the center of the graph, the first landmark is chosen near the median of the graph.

**Avoid** Suppose a set $\mathcal{L}$ of landmarks has already been selected. To add a new landmark to this set, the shortest path tree $T_s$, rooted at some node $s$, is calculated. The weight of node $v$ is defined as the difference between $d(s, v)$ and the lower bound for $d(s, v)$ given by $h(s, v)$ (see (6.6)). The weight of the node is a measure for the distance estimates at that time. The size of node $v$ depends on $T_v$, the subtree of $T_s$ rooted at $v$. The size of $v$ is 0 if $T_v$ contains a landmark. In all other cases, the size of $v$ is the sum of the weights of all nodes in $T_v$. Let $w$ be the node with maximum size. Traverse $T_w$, starting at $w$ and always following the child with largest size, until a leaf is reached. This leaf is the new landmark that must be added to $\mathcal{L}$.

This strategy is called avoid, because it tries to identify regions of the graph that are not so well covered by avoiding existing landmarks. There are no
landmarks in the subtrees of any node in $T_w$. By adding a leaf of $T_w$ to the set of landmarks, one tries to improve the coverage. Node $s$ can be taken uniformly at random from $V$, but better results are found if the selection is biased towards nodes far away from $L$.

6.3.3.1 Local search

As stated in [96], a downside of selecting landmarks by using a constructive heuristic as described above is that some earlier selected landmarks might be of limited use once others are selected. A standard local search technique, to improve the quality of the set of selected landmarks, is to repeatedly remove a landmark from the set and add a landmark to the set. There are several ways to determine which landmark should be added to the set:

- In [89], it is proposed to choose the landmark that lies farthest from the landmarks already selected. In fact, the farthest selection strategy described above is continued. Again, the distance to the set of landmarks can be measured either in arc costs or in number of arcs.
- Based on a sample set of origins and destinations, the score of a new landmark $L \notin L$ can be measured as follows: Initially, the score of landmark $L$ is made zero. For any pair $(s, t)$ in the sample set, the value $h(s, t)$ (see (6.6)) is compared to $h_L(s, t)$ (see (6.5)). If $h_L(s, t) > h(s, t)$, then the score of landmark $L$ is increased by $h_L(s, t) - h(s, t)$. The landmark with the highest score is added to the set of selected landmarks.

In [96], the reduced arc costs (see Definition 3.9) $c_f(v, w)$ are exploited where the potential function $f$ is taken as $h_L^*$. If the reduced arc cost $c_v + h_L^*(v) - h_L^*(w) = 0$, then arc $(v, w)$ is said to be covered by landmark $L$. The idea is to select landmarks in such a way that the number of arcs covered by at least one landmark, is maximized. Based on an initial set of landmarks (selected by the avoid strategy described above), a local search method is proposed to select a subset from these candidate landmarks as landmark set. This heuristic improvement method is called maxcover.

In Figure 6.9, the selected landmarks are shown for two different selection strategies. The left part shows the landmarks selected by the farthest selection rule, where the distance is measured in number of arcs. The right part shows the result of the maxcover heuristic (which is a heuristic improvement of the avoid selection rule). It is clear that many landmarks are lying on the border of the graph, especially when the maxcover strategy is used.

6.3.3.2 Active landmarks

Let $L$ be a set of landmarks. Evaluating the landmark estimator (6.6) might be computationally expensive if $|L|$ is large. In [89], it is proposed to use only a subset
6.3 Dynamic graph preprocessing

Figure 6.9: Landmark sets. Based on the farthest selection strategy (left). Based on maxcover (right).

$L' \subset L$, called the active landmark set. At the start of an $s$-$t$ shortest path calculation, it is determined which set of active landmarks $L'$ should be used. The landmarks in $L$ are ordered by non-increasing lower bounds $h_L(s, t)$ (see (6.5)). Then, the first $k$ landmarks of this ordered set are chosen as active landmarks. Typically, $|L'| = 4$ while $|L| = 16$ or $|L| = 64$.

Selecting active landmarks this way will provide good lower bounds for the nodes in the neighborhood of $s$ and $t$, but they might be less fruitful for nodes with approximately the same distance to $s$ and $t$. In [121], we proposed another method for selecting four active landmarks:

- $L_1 \leftarrow \text{arg max} \{h_L(s) | L \in \mathcal{L}\}$,
- $L_2 \leftarrow \text{arg min} \{h_L^+(s) | L \in \mathcal{L}\} \setminus \{L_1\}$,
- $L_3 \leftarrow \text{arg min} \{h_L^+(t) | L \in \mathcal{L}\} \setminus \{L_1, L_2\}$,
- $L_4 \leftarrow \text{arg min} \{h_L^-(s) + h_L^-(t) | L \in \mathcal{L}\} \setminus \{L_1, L_2, L_3\}$.

This way, $L_1$ maximizes the lower bound for $d(s, t)$, $L_2$ lies nearby $s$, $L_3$ lies nearby $t$ and $L_4$ is somewhere in the middle of $s$ and $t$.

Goldberg and Werneck [96] suggested the use of a dynamic set of active landmarks. Initially, $L' = \{L_1, L_2\}$ where $L_1 \leftarrow \text{arg max} \{h_L^+(s) - h_L^-(t) | L \in \mathcal{L}\}$ and $L_2 \leftarrow \text{arg max} \{h_L^-(t) - h_L^-(s) | L \in \mathcal{L}\}$. Let $b$ be the lower bound for $d(s, t)$ based on the initial landmarks $L_1$ and $L_2$. Periodically, a new landmark might be added to the set of active landmarks $L'$, until the algorithm terminates or $|L'|$ reaches a certain upper bound (six in their experiments). An attempt to update $L'$ occurs whenever a search (in any direction) selects a pivot node $v$ with a heuristic estimate $h_v$ that is smaller than a certain checkpoint. For the $i^{th}$ checkpoint, they use $b(10 - i)/10$. 

\[ L' \subset L, \text{ called the active landmark set. At the start of an s-t shortest path calculation, it is determined which set of active landmarks L' should be used. The landmarks in L are ordered by non-increasing lower bounds h_L(s, t) (see (6.5)). Then, the first k landmarks of this ordered set are chosen as active landmarks. Typically, |L'| = 4 while |L| = 16 or |L| = 64. Selecting active landmarks this way will provide good lower bounds for the nodes in the neighborhood of s and t, but they might be less fruitful for nodes with approximately the same distance to s and t. In [121], we proposed another method for selecting four active landmarks:

- L_1 \leftarrow \text{arg max} \{h_L(s) | L \in \mathcal{L}\},
- L_2 \leftarrow \text{arg min} \{h_L^+(s) | L \in \mathcal{L}\} \setminus \{L_1\},
- L_3 \leftarrow \text{arg min} \{h_L^+(t) | L \in \mathcal{L}\} \setminus \{L_1, L_2\},
- L_4 \leftarrow \text{arg min} \{h_L^-(s) + h_L^-(t) | L \in \mathcal{L}\} \setminus \{L_1, L_2, L_3\}.

This way, L_1 maximizes the lower bound for d(s, t), L_2 lies nearby s, L_3 lies nearby t and L_4 is somewhere in the middle of s and t.

Goldberg and Werneck [96] suggested the use of a dynamic set of active landmarks. Initially, L' = {L_1, L_2} where L_1 \leftarrow \text{arg max} \{h_L^+(s) - h_L^-(t) | L \in \mathcal{L}\} and L_2 \leftarrow \text{arg max} \{h_L^-(t) - h_L^-(s) | L \in \mathcal{L}\}. Let b be the lower bound for d(s, t) based on the initial landmarks L_1 and L_2. Periodically, a new landmark might be added to the set of active landmarks L', until the algorithm terminates or |L'| reaches a certain upper bound (six in their experiments). An attempt to update L' occurs whenever a search (in any direction) selects a pivot node v with a heuristic estimate h_v that is smaller than a certain checkpoint. For the i^{th} checkpoint, they use b(10 - i)/10. \]
Thus, the first attempt to add a new landmark to the active landmark set is made, when $h_v$ reaches 90% of the initial lower bound $b$ for $d(s, t)$. The second attempt is made when the lower bound reaches 80% of $b$, and so on. Furthermore, it is required to perform at least a certain number of iterations (100 for example), between each two consecutive attempts to add a landmark. If this minimum amount of in-between iterations is not used, it may happen that for relative short $s$-$t$ distances, too many attempts are made. At each attempt, the algorithm verifies whether or not it is possible to improve the heuristic estimate $h_v$ of $v$ by using a landmark $L \in \mathcal{L}\setminus\mathcal{L}'$ with at least a factor $1 + \varepsilon$ (they used $\varepsilon = 0.01$). If so, the landmark $L$ is added to the set of active landmarks $\mathcal{L}'$. Once $\mathcal{L}'$ is updated, the heuristic estimates $h^s_v (v \in \mathcal{Q}^s)$ and $h^t_v (v \in \mathcal{Q}^t)$ have to be recalculated.

### 6.3.3.3 Reducing memory usage

The number of landmarks $|\mathcal{L}|$ is most commonly physically bounded by the amount of memory available. On our test graph (see Section 6.1.1), we have $|\mathcal{V}_{\text{car}}| = 8,678,011$ and thus adding one landmark results in storing $2 \times 8,678,011$ distances. If a distance is stored as a 32 bit value, we need about 66 MB of memory to add one landmark.

There are several ways to reduce the memory needed for one landmark at the cost of a weaker lower bound. In this section, we mention two methods to save memory.

We observed earlier [121] that the relative difference between the cost $d(s, t)$ of the shortest path from $s$ to $t$ and the cost $d(t, s)$ of the shortest path from $t$ to $s$ is marginal, especially with a relatively large distance between $s$ and $t$. To reduce the amount of memory needed, we do not store the distance $h^+_L(v)$ from node $v$ to landmark $L$ and the distance $h^-_L(v)$ from landmark $L$ to node $v$, but only an undirected distance $h_L(v)$. For each landmark $L$, we determine the shortest path tree on the graph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ with the modified costs $\tilde{c}_{vw} = \min(c_{vw}, c_{wv})$. Note that $c_{wv} = \infty$ if $(w, v) \notin \mathcal{A}$. For each node $v \in \mathcal{V}$, we store the distance $h_L(v)$ from root $L$ to node $v$ in this shortest path tree.

By the triangle inequality, $h_L(v) - h_L(w)$ is a lower bound for $\tilde{d}(v, w)$, which in turn is the length of a shortest path from $v$ to $w$ in the graph with modified arc costs $\tilde{c}_{uw}$. Furthermore, it is clear that $\tilde{d}(v, w) \leq d(v, w)$, the shortest path from $v$ to $w$ in the graph with original arc costs $c_{uw}$. Thus:

$$h_L(v) - h_L(w) \leq d(v, w). \quad (6.7)$$

The landmark estimator we introduce is:

$$h'_v = \max \{h_L(v) - h_L(t) | L \in \mathcal{L}\}. \quad (6.8)$$

Since (6.7) shows that the undirected lower bound is consistent (see Definition 5.5), the undirected landmark estimator (6.8) is also consistent (see Theorem 5.8).
Using the undirected landmark-estimator (6.8), we reduce the memory consumption with a factor 2.

In [91] and [52], methods are described to restrict the landmark distances to a subset of $\mathcal{V}$. For example, one can choose landmarks only from $\mathcal{V}^0$, which is the set of core nodes. Recall that the set of core nodes can be determined either by static preprocessing (Section 6.2) or by graph contraction (Section 6.3.1). Instead of storing the distances to and from a landmark for any node $v \in \mathcal{V}$, we only store distances to and from the landmark for all core nodes $v \in \mathcal{V}^0$.

During the $A^*$ algorithm, we need to calculate lower bounds for the distance between a node $v$ and the destination node $t$. Both nodes $v$ and $t$ are not necessarily core nodes. Since we only store distances from and to a core node for each landmark, the landmark estimator (6.6) cannot be used straightforwardly. To overcome this problem, we need the concept of proxy nodes.

**Definition 6.9** Let $v \in \mathcal{V}\setminus\mathcal{V}^0$ be a non-core node. The proxy node $v' \in \mathcal{V}^0$ is a core node for which $d(v, v')$ is minimal.

It should be noted that, for any non-core node $v$, the proxy node $v' \in \mathcal{V}^0$ and the distance $d(v, v')$ can be stored during a preprocessing step.

In Figure 6.10 [13], it is illustrated how proxy nodes can be used to determine a lower bound on the distance from a core node $v$ to a non-core node $t$. From this figure, it can readily be seen that:

$$
\begin{align*}
    d(v, t') &\leq d(v, t) + d(t, t'), \\
    h^+(v) &\leq d(v, t') + h^+(t'), \\
    h^-(t') &\leq d(v, t') + h^-(v).
\end{align*}
$$

**Figure 6.10:** Computing a lower bound for $d(v, t)$ with $v \in \mathcal{V}^0$ and $t \notin \mathcal{V}^0$, using only landmark distances to and from core nodes.

Hence, a lower bound $h''(v, t)$ for $d(v, t)$ is:

$$
h''(v, t) = \max \{ h^+(v) - h^+(t') - d(t, t'), h^-(t') - h^-(v) - d(t, t') | L \in \mathcal{L} \}. 
$$

We can rewrite (6.9) as: $h''(v, t) = \max \{ h_L(v) | L \in \mathcal{L} \} - d(t, t')$. Here, $h_L(v)$ is defined according to (6.5). Since $d(t, t')$ is constant, the landmark estimator (6.9)
can be read as the original landmark estimator (6.6) minus this constant. Thus, this landmark estimator is consistent, with respect to core nodes.

For any non-core node, the landmark estimate is not defined. We define $h''(v, t) = 0$ for all nodes $v \in V \setminus V_0$. As seen in Section 5.2.2, the resulting estimator is optimistic but inconsistent.

\[
h''(v, t) = \begin{cases} 
\max \{ h_L(v) | L \in \mathcal{L} \} - d(t, t'), & \forall v \in V_0, \\
0, & \text{otherwise}.
\end{cases}
\]  

(6.10)

We can incorporate (6.10) into the $A^*$ algorithm (Algorithm 5.1), but the resulting algorithm will be label-correcting. Furthermore, we may incorporate (6.10) in a bidirectional algorithm that makes use of scalar projections (see Section 5.3.4). Other bidirectional algorithms, using either a balanced or a symmetric approach, cannot be used straightforwardly since they require consistent heuristic estimators.

In [13], a two-phase algorithm CALT ("Core-ALT") is presented (see Algorithm 6.4). The first phase runs each direction separately. Since the cardinality of each partition of the node set is rather small, it neither makes sense to run this phase for both directions in parallel, nor to alternate between forward and backward search. In the first phase, only the outgoing arcs of non-core nodes are scanned. Core nodes are added to a so-called entry set. In the forward search, such pivot nodes are stored in $S$ and in the backward search in $T$. It is possible that a node $v$ is scanned from both sides during the first phase. Thus, the meeting node of the tentative shortest path is stored in $u$. The first phase is described in Algorithm 6.5.

The second phase is initialized by re-initializing $Q^*$ to become the set of forward entry nodes $S$ and $Q^t$ to become the set of backward entry nodes $T$. The second phase alternates between forward and backward searches until a certain stop condition holds. For details about alternate rules and stop conditions, see Sections 4.2.2 and 4.2.1. Since in the second phase only core nodes are taken into account (see Algorithm 6.6), the estimator (6.9) is consistent. Thus, it is possible to incorporate it in either a balanced (see Section 5.3.2) or in a symmetric way (see Section 5.3.3).

Both phases (Algorithms 6.5 and 6.6) are presented here as label-setting phases. It is easy to adapt these algorithms to label-correcting variants, as described in Section 4.3.

Landmark estimators can easily be combined with node reaches (see Section 6.3.2).
Algorithm 6.4: CALT \((G^s, G^t, s, t, h^s, h^t)\)

1: \(u \leftarrow \text{NIL}\)
2: CALT-PHASE-I\((G^s, Q^s, s, S)\)
3: CALT-PHASE-I\((G^t, Q^t, t, T)\)
4: \(Q^s \leftarrow S\)
5: \(Q^t \leftarrow T\)
6: repeat
7: \text{if a certain selection rule holds then} \quad \text{Iterate (CALT)}\((G^s, Q^s, h^s)\)
8: \text{else} \quad \text{Iterate (CALT)}\((G^t, Q^t, h^t)\)
9: until a certain stop condition holds

Algorithm 6.5: CALT-PHASE-I \((G, Q, s, S)\)

1: \text{INITIALIZE-SINGLE-SOURCE}(G, s, Q)
2: \(S \leftarrow \emptyset\)
3: repeat
4: \text{Let } v \leftarrow \arg\min\{d_v | v \in Q\}
5: \(Q \leftarrow Q - \{v\}\)
6: \text{if } u = \text{NIL} \text{ or } d_v^u + d_v^t < d_u^s + d_u^t \text{ then} \quad \text{\(u \leftarrow v\)}
7: \text{if } v \in V^0 \text{ then} \quad \text{\(S \leftarrow S \cup \{v\}\)}
8: \text{\text{else} \quad foreach } (v, w) \in \delta^+(v) \text{ do} \quad \text{LABEL-UPDATE}(v, w, Q)
9: until \(Q = \emptyset\)

Algorithm 6.6: Iterate (CALT)\((G, Q)\)

1: \text{Let } v \leftarrow \arg\min\{d_v | v \in Q\}
2: \(Q \leftarrow Q - \{v\}\)
3: \text{foreach } (v, w) \in \delta^+(v) \text{ do}
4: \text{if } w \in V^0 \text{ then} \quad \text{LABEL-UPDATE}(v, w, Q)
5: \text{if } w \in \bar{R} \text{ then}
6: \text{if } u = \text{NIL} \text{ or } d_w + \tilde{d}_w < d_u + \tilde{d}_u \text{ then} \quad \text{\(u \leftarrow w\)}
6.4 Contraction Hierarchies

In 2008, Geisberger et al. [79, 81] introduced a preprocessing technique called Contraction Hierarchies. This preprocessing technique is in fact a dynamic preprocessing technique, closely related to graph contraction (see Section 6.3.1). Geisberger presented an algorithm to solve the SSSD-SPP that utilizes the results of the preprocessing step. Contraction Hierarchies are also used in a very fast, the current state-of-the-art, algorithm called Hub Based Labeling [1], which differs from all other algorithms described in this thesis. Instead of being a specific implementation of the Generic Shortest Path algorithm (Algorithm 3.4), Hub Based Labeling is more a lookup algorithm. Therefore, we decided to write about Contraction Hierarchies and Hub Based Labeling in a separate section, instead of adding it to the dynamic preprocessing section (Section 6.3).

It turns out that Contraction Hierarchies closely resembles the ideas of directional path-consistency, introduced in 1988 by Dechter and Pearl [47]. In Appendix C, we describe the theory about directional path-consistency and the similarities between directional path-consistency and Contraction Hierarchies.

This section is organized as follows. In Section 6.4.1, we describe the Contraction Hierarchies preprocessing algorithm. Section 6.4.2 describes Geisberger’s algorithm to solve the SSSD-SPP based on the Contraction Hierarchies. The performance of that algorithm is strongly influenced by the node ordering used during the creation of the Contraction Hierarchies (during the preprocessing). In Section 6.4.3, several heuristics to find a node ordering that leads to a fast SSSD-algorithm are mentioned. Finally, in Section 6.4.4 we describe the Hub Based Label algorithm of Abraham et al. in detail.

6.4.1 Construction of Contraction Hierarchies

Originally, Contraction Hierarchies are described as a special case of another preprocessing technique, called Highway Node Routing [176]. Highway Node Routing makes use of a hierarchy of levels $\mathcal{L} = \{1, 2, \ldots, L\}$, similar to Highway Hierarchies ([170, 171]). Each node $v$ is assigned to such a level. Let $\ell : V \to \mathcal{L}$ be the mapping from the nodes to the levels. During the creation of the hierarchy, it is ensured for each pair of nodes $v$ and $w$ in the original graph that in the resulting graph a shortest path exists that contains only nodes $u$ such that $\ell(u) \geq \min\{\ell(v), \ell(w)\}$. Therefore, each time a new (higher) level is added to the hierarchy, the nodes of the current level are bypassed (see Definition 6.8).

The number of levels used in Highway Hierarchies and Highway Node Routing, is limited. In contrast, Contraction Hierarchies assign a separate level to each node. Algorithm 6.7 describes the way how a graph is preprocessed in order to create a contraction hierarchy. This algorithm uses an ordering ‘$<$’ on $V$. This ordering on the nodes in $V$ is commonly defined as $v_i < v_j$ if and only if $i < j$. 
The main “for-loop” (line 1) of Algorithm 6.7 selects the next node \( v_k \) to be bypassed. Node \( v_k \) is bypassed in a similar way as in Definition 6.8, except for the check in line 6. Geisberger [79] introduced the concept of a witness path, which can be used to implement this check.

**Definition 6.10** A \( v \)-w path \( P \neq (v, (v, u), (u, w), w) \) with \( \ell(P) \leq c_{vu} + c_{uw} \) is called a witness path for the triple \((v, u, w)\). Such a path ensures that either \((v, (v, u), (u, w), w)\) is not a shortest \( v \)-w path or it is not the only shortest \( v \)-w path.

**Algorithm 6.7: Contraction Hierarchies**

```plaintext
1: for \( k = 1 \) to \(|V|\) do
2:     foreach \((v_i, v_k) \in \delta^-(v_k)\) do
3:         if \( v_k < v_i \) then
4:             foreach \((v_k, v_j) \in \delta^+(v_k)\) do
5:                 if \( v_k < v_j \) and \( v_i \neq v_j \) then
6:                     if \( v_k \) “might be” on each shortest \( v_i \)-v_j path then
7:                         if \((v_i, v_j) \in A\) then
8:                             \( c_{v_i, v_j} \leftarrow \min\{c_{v_i, v_j}, c_{v_i, v_k} + c_{v_k, v_j}\}\)
9:                         else
10:                            \( A \leftarrow A \cup \{(v_i, v_j)\}\)
11:                            \( c_{v_i, v_j} \leftarrow c_{v_i, v_k} + c_{v_k, v_j}\)
```

If, during iteration \( k \) in Algorithm 6.7, a witness path is found for the triple \( v_i, v_k, v_j \) that contains only nodes \( v > v_k \), neither the shortcut \((v_i, v_j)\) is added, nor the arc cost of the possible existing arc \((v_i, v_j)\) is updated.

We define the so-called upward and downward graph.

**Definition 6.11** Let \( G = (V, A) \) be a digraph and ‘\(<\)’ be an ordering on the nodes in \( V \). The upward graph \( G^+ = (V, A^+) \) is defined by \( A^+ = \{(v_i, v_j) | (v_i, v_j) \in A, v_i < v_j\} \) and the downward graph \( G^- = (V, A^-) \) is defined by \( A^- = \{(v_i, v_j) | (v_i, v_j) \in A, v_i > v_j\} \).

Definition 6.11 resembles the idea of Yen [203] about partitioning the arc set \( A \), which we mentioned in Section 3.4. Yen used such a partitioning of the arc set to improve Bellman-Ford-Moore (Algorithm 3.7). In each main iteration of Algorithm 3.7, he first iterates over \( v_1, v_2, \ldots, v_{|V|} \) and calls the LABEL-UPDATE subroutine only for arcs in \( A_1 \). Next, he iterates over \( v_{|V|}, v_{|V|-1}, \ldots, v_1 \) and calls the LABEL-UPDATE subroutine only for arcs in \( A_4 \).

**Proposition 6.12** Let \( G = (V, A) \) be a digraph and ‘\(<\)’ an ordering on the nodes. Let \( G^+_1 \) be the upward graph and \( G^-_1 \) be the downward graph based on the graph \( G^* \) that results after the execution of the Contraction Hierarchies \((G, <)\) algorithm. Let
$P_{v_i,y}$ be a shortest path in $G$. There exists a path $P' = (v_i = u_0, (u_0, u_1), \ldots, u_p, \ldots, (u_{k-1}, u_k), u_k = v_j)$ in $G'$, which has the same length as $P$ where $(u_i, u_{i+1}) \in A^+_t$ for $0 \leq i \leq p - 1$ and $(u_i, u_{i+1}) \in A^+_s$ for $p \leq i \leq k - 1$.

**Proof:** No arcs are removed from $G$, during the execution of Algorithm 6.7. Furthermore, any new arc $(v, w)$ that is added, has the length of a $v$-$w$ path that lies entirely in $G$. Hence, $G'$ contains no shorter paths than $G$ and there exists a shortest $v_s$-$v_t$ path in $G'$ with length $\ell(P)$. We have to prove that a $v_s$-$v_t$ path of the form $P'$ exists. Let $P^* = (v_s = u_0, (u_0, u_1), u_1, \ldots, (u_{k-1}, u_k), u_k = v_j)$ be a shortest path in $G'$ that does not have the form of $P'$. Let $u_p (1 < p < k)$ be the first node in $P^*$, such that $u_{p-1} > u_p$ and $u_{p+1} > u_p$. Such a node $u_p$ exists, since $P^*$ is not of the form of $P'$. Both arcs $(u_{p-1}, u_p)$ and $(u_p, u_{p+1})$ already existed at the iteration of Algorithm 6.7, in which $u_p$ was bypassed. Hence, there exist a witness path $P_{u_{p-1}, u_{p+1}}$ for the triple $u_{p-1}, u_p, u_{p+1}$ with $\ell(P_{u_{p-1}, u_{p+1}}) = c_{u_{p-1}, u_p} + c_{u_p, u_{p+1}}$. $G'$ contains the path $P_{u_{p-1}, u_{p+1}} = (u_{p-1}, (u_p, u_{p+1}), u_{p+1})$ with the same length. Let $\hat{P}$ be the path that results if the subpath $((u_{p-1}, u_p), u_p, (u_p, u_{p+1}))$ of $P^*$ is replaced by $P_{u_{p-1}, u_{p+1}}$. $\hat{P}$ lies in $G'$ and has the same length as $P^*$. The process of replacing a subpath by another one is continued until the resulting path has the form of $P'$.

The process will eventually end, since the number of nodes in any path is bounded and with each replacement, the possible position $p$ of a violation (that is, a node $u_p$ such that $u_{p-1} > u_p$ and $u_{p+1} > u_p$) is increased by one. $\square$

The question is how to efficiently determine whether or not there exists a witness path for a given triple of nodes. A multiple-source, multiple-destination shortest path problem (MSMD-SPP), has to be solved at the start of each main iteration, where the set of source nodes $S$ consists of the nodes $v_i > v_k$ such that $(v_i, v_k) \in A$ and the set of destination nodes $T$ consists of the nodes $v_j > v_k$ such that $(v_k, v_j) \in A$. In [81], an implementation to solve this MSMD-SPP is mentioned. It is based on a bidirectional method as described in [122]. First, for any destination node $v_j \in T$, one backward iteration is performed. In fact, for each $u > v_k$, a bucket is maintained. The bucket of node $u$ contains all nodes $v_j \in T$ with $(u, v_j) \in \delta^-(v_j)$. Next, a shortest path tree is grown (in the forward direction) for any source node $v_i \in S$, omitting all nodes $u \leq v_k$. Each time a node $u$ is added to the shortest path tree, the bucket of node $u$ is scanned. For any node $v_i$ in the bucket, a path with length $d_u + c_{u, v_j}$ is found. If that path is a witness path, no shortcut is needed between $v_i$ and $v_j$.

The growth of the shortest path tree rooted at $v_i$, can be stopped as soon as for each node $u$ in the candidate set $Q$, the following inequality holds:

$$d_u > c_{v_i, v_k} + \max_{v_j \in T}\{c_{v_k, v_j} - \min_{(w, v_j) \in A, w > v_k} c_{w, v_j}\}.$$

Determining exact shortest paths between any pair of nodes $v_i \in S$ and $v_j \in T$ might be expensive. In [81], two methods are proposed to limit the time for finding witness paths:
6.4 Contraction Hierarchies

- Limit the number of iterations performed to grow the shortest path tree.
- Limit the number of edges on each path considered.

Limiting the search for a witness path might lead to not finding an existing witness path at all. This does not influence the correctness of the algorithm. However, a shortcut is added unnecessarily.

Note that for each shortcut \((v, w)\) added to \(G\), nodes \(v\) and \(w\) are higher than all internal nodes of the according shortest \(P_{vw}\) path. Abraham et al. [2] use this observation in the following way. Instead of bypassing all nodes one-by-one, the shortcuts are determined in two phases. In the first phase, the shortcuts \((v, w)\) with \(w > v\) are determined by growing a shortest path tree rooted at \(s\) for each node \(s \in V\), using \textsc{Dijkstra}. Each time a node \(v > s\) is chosen as pivot node, it is checked whether or not \(v\) is the first node higher than \(s\) on the \(s-v\) path. If so, the shortcut \((s, v)\) is added. \textsc{Dijkstra} can be stopped as soon as each node \(v\) in the candidate set \(Q\) has a parent \(u > s\). In the second phase, the shortcuts \((v, w)\) with \(v < w\) are determined in a similar way.

### 6.4.2 Solving the SSSD-SPP with Contraction Hierarchies

Geisberger adapted the bidirectional shortest path algorithm of Schultes [175], as described in Algorithm 6.8. This algorithm can run on the graph that results after the creation of Contraction Hierarchies (Algorithm 6.7).

#### Algorithm 6.8: Shortest Path (CH) \((G_\uparrow, G_\downarrow, s, t)\)

1. \(u \leftarrow \text{NIL}\)
2. \text{INITIALIZE-SINGLE-SOURCE}(G\_\uparrow, s, Q^s)
3. \text{INITIALIZE-SINGLE-SOURCE}(G\_\downarrow, t, Q^t)
4. repeat
   5. if a certain selection rule holds then
      6. \text{ITERATE}(G\_\uparrow, Q^s)
   7. else
      8. \text{ITERATE}(G\_\downarrow, Q^t)
5. until a certain stop condition holds

Originally, the algorithm alternates between forward and backward searches. In Section 4.2.2, other rules for selecting the direction of search are possible. These rules can be used in Algorithm 6.8 as well, except for the pure unidirectional search. Clearly, it might happen that one of the search directions is stopped (since its candidate set becomes empty), while the other direction has to be explored further.

The stop condition in its most basic form is: \(Q^s = Q^t = \emptyset\). As soon as \(u \neq \text{NIL}\) however, we have a tentative shortest path with length \(d^s_u + d^t_u\). As soon as each node \(v \in Q^s\) has a label \(d^s_v \geq d^s_u + d^t_u\), the forward search can be stopped. A same argument holds for the backward search.
Proposition 6.13 Algorithm 6.8 correctly determines a shortest s-t path if $G$ is the result of Algorithm 6.7.

Proof: Since $G$ is the result of Algorithm 6.7, there exists a shortest s-t path $P$ of the form $P = (v_s = u_0, (u_0, u_1), \ldots, (u_{p-1}, u_k), u_k = t)$ where $(u_i, u_{i+1}) \in A_i$ for $0 \leq i < p - 1$ and $(u_i, u_{i+1}) \in A_k$ for $p \leq i < k - 1$ (see Proposition 6.12). The forward search will eventually find the path $P = (v_s = u_0, (u_0, u_1), \ldots, u_p)$ and the path $P = (u_p, \ldots, (u_{k-1}, u_k), u_k = t)$ will eventually be determined by the backward search.

Notice that if the abovementioned stop condition is used, it might happen that another shortest path is found earlier.

A well-known speedup technique in Highway Hierarchies, called stall-on-demand [175], can be used with Contraction Hierarchies as well. As soon as a pivot node $v$ is selected in the forward search, the arcs $(w, v) \in G_i$ (hence, $w > v$) are inspected. If $d^*_w + c_{wv} < d^*_v$, node $v$ is not scanned but stalled with stalling distance $d^*_w + c_{wv}$. A stalled node $v$ is woken up if an s-v path in $G_i$ is found that is shorter than the current value of $d^*_v$. However, node $v$ might be stalled again, immediately after waking up. The stall-on-demand technique can be applied to the backward search as well.

Contraction hierarchies can easily be used to solve the MSMD-SPP. Geisberger [80] adapted contraction hierarchies also in timetabled networks (e.g. public transport networks).

6.4.3 Determining a node ordering

Geisberger [79, 81] did a lot of research on finding a good node ordering for his Contraction Hierarchies. He used a non-monotonic priority queue to determine the node to be bypassed in the current iteration.

Definition 6.14 A non-monotonic priority queue is an ADT that supports the following operations:

- **Create.** Return a new, empty priority queue.
- **Insert**$(v, k)$. Add an item $v$ with a predefined key $k$ to the priority queue.
- **Find-Min**. Return an item $v$ with the smallest key among all items in the priority queue.
- **Delete-Min**$(v)$. Delete an item $v$ with the smallest key among all items in the priority queue and return $v$.
- **Decrease-Key**$(v, k)$. Assign the new (lower) key $k$ to item $v$.
- **Increase-Key**$(v, k)$. Assign the new (higher) key $k$ to item $v$. 
A non-monotonic priority queue therefore supports the same operations as a priority queue (see Appendix B) as well as the \textsc{Increase-Key} operation.

Geisberger introduced the concept of \textit{lazy updates}. In a priority queue, the top node (the node that is selected with the \textsc{Delete-Min} operation) has the smallest key among all nodes in the priority queue. Each time the key of a node has to be increased, this can be postponed. As soon as node $v$ is returned by the \textsc{Delete-Min} operation, the key $k$ of node $v$ is determined. If this key is larger than the key of the next top node of the priority queue, then node $v$ is simply re-inserted with $\textsc{Insert}(v, k)$ and the new top node of the queue is selected. The \textsc{Delete-Min} operation stops as soon as a node $v$ is found for which the key is smaller than, or equal to, the key of the new top node of the priority queue (or $v$ was the only node in the queue). He observed that periodically recomputing all keys, and thus rebuilding the priority queue, improved the performance when a priority queue with lazy updates is used.

![Figure 6.11: A ‘dead valley’.](image)

The key used in the non-monotonic priority queue can be any linear combination of several terms. The most important term is \textit{arc difference}. The arc difference is defined by the number of shortcuts that are added during the bypassing of node $v$ minus the number of arcs that are incident to $v$ (that is, $|\delta^+(v)| + |\delta^-(v)|$). Clearly, the arc difference can be negative. Using only the arc difference, the resulting single-source single-destination shortest path algorithm can be quite slow. Suppose the original graph $G = (V, A)$ contains a ‘dead valley’, as presented in Figure 6.11. If the nodes are chosen in the order $v_1, v_2, \ldots, v_k$, then a shortest path from node $v_1$ will always visit all arcs $(v_i, v_i + 1)$ for $1 \leq i < k$.

It seems to be a good idea to bypass nodes everywhere in the graph in a uniform way, rather than to keep contracting nodes in a small region. Geisberger mentioned two heuristic terms that stimulate a uniform node ordering:

- \textit{Deleted neighbors}. The number of neighbors that have already been bypassed, is counted for each node. This, of course, includes neighbors that are reached via shortcuts.
• **Voronoi regions.** The Voronoi-Region $R(v)$ is defined as the set of already bypassed nodes that are closer to $v$ than to any other node of the remaining graph. The square root $\sqrt{R(v)}$ is used as a term in the linear function to determine the key. The idea is that bypassing nodes with small Voronoi regions leads to a uniform spread of the node ordering. As soon as node $v$ is contracted, its neighbors will 'consume' all nodes in $R(v)$. The (re-)calculation of the Voronoi regions is done by growing a partial shortest path tree.

It should be noted that both terms *Deleted neighbors* and *Voronoi regions* can only increase. Hence, the idea of lazy updates can be used. That is, the recalculation of the actual values for these terms can be postponed until the node is selected by the **DELETE-MIN** procedure.

### 6.4.4 Hub Based Labeling

Abraham et al. use Contraction Hierarchies in a so-called Labeling algorithm. Before we describe their algorithm we first define distance labeling schemes more formally. We follow the definitions of [78].

**Definition 6.15** Let $G = (V, E)$ be an undirected graph. A node-labeling for the graph $G$ is a function $L$ that assigns a label $L(v)$ to each node $v \in V$.

**Definition 6.16** A distance decoder is a function $h$ responsible for distance computation. Given two labels $L(v)$ and $L(w)$, it returns $h(L(v), L(w))$.

**Definition 6.17** Let $L$ be a node-labeling on a graph $G = (V, E)$ and let $h$ be a distance decoder. The pair $(L, h)$ is a distance labeling if $h(L(v), L(w)) = d(v, w)$ for any pair of nodes $v, w \in V$.

From Definition 6.16, it is clear that the distance decoder $h$ itself does not depend on the graph $G$. In contrast, the labels contain some information that can be precomputed by considering $G$. We now extend these definitions for directed graphs:

**Definition 6.18** Let $G = (V, A)$ be a digraph. A directed node-labeling for the graph $G$ is a function $L$ that assigns a forward label $L_f(v)$ and a backward label $L_b(v)$ to each node $v \in V$.

**Definition 6.19** A directed distance decoder is a function $h$, responsible for distance computation. Given a forward label $L_f(v)$ and a backward label $L_b(w)$, it returns $h(L_f(v), L_b(w))$.

**Definition 6.20** Let $L$ be a directed node-labeling on a digraph $G = (V, A)$ and let $h$ be a directed distance decoder. The pair $(L, h)$ is a directed distance labeling if $h(L_f(v), L_b(w)) = d(v, w)$ for any pair of nodes $v, w \in V$.

Abraham et al. defined the cover property for a direct node-labeling.
Definition 6.21 Let \( G = (V, A) \) be a digraph and \( L \) be a directed node-labeling such that \( L_f(v) \) is a set of pairs \( \langle w, d(v, w) \rangle \) with \( w \in V \). Thus, the forward label \( L_f(v) \) contains several nodes and the shortest path distances from node \( v \) to the nodes in \( L_f(v) \). The backward label \( L_b(v) \) is a set of pairs \( \langle u, d(u, v) \rangle \) with \( u \in V \). Such a directed node-labeling has the cover property if, for every pair of distinct nodes \( s, t \in V \), \( L_f(s) \cap L_b(t) \) contains a node \( u \) that is on a shortest \( s-t \) path.

If a directed node labeling \( L \) has the cover property, it is immediately clear how the directed distance decode \( h \) can be evaluated:

\[
h(L_f(v), L_b(w)) \leftarrow \min\{d(v, u) + d(u, w) | u \in L_f(v) \cap L_b(w)\}
\] (6.11)

Suppose that in each label (either forward or backward) of \( L \), the set of pairs are sorted according to a certain node ordering \( ' < ' \), then the evaluation of (6.11) can be done in \( O(|L_f(v)| + |L_b(w)|) \) time, as shown in Algorithm 6.9.

**Algorithm 6.9: Hub Based Labeling(\( s, t, L, < \))**

**Require:** \( L \) is a directed node labeling that has the cover property and \( ' < ' \) is a node ordering. The set of pairs in each label of \( L \) are sorted on this ordering.

1: \( d \leftarrow \infty \)
2: \( i \leftarrow 0 \)
3: \( j \leftarrow 0 \)
4: while \( i < |L_f(s)| \) and \( j < |L_b(t)| \) do
5: Let \( v_i \) be the \( i \)th node in label \( L_f(s) \)
6: Let \( w_j \) be the \( j \)th node in label \( L_b(t) \)
7: if \( v_i = w_j \) then
8: \( d \leftarrow \min\{d, d(s, v_i) + d(w_j, t)\} \)
9: \( i \leftarrow i + 1 \)
10: \( j \leftarrow j + 1 \)
11: else
12: if \( v_i < w_j \) then
13: \( i \leftarrow i + 1 \)
14: else
15: \( j \leftarrow j + 1 \)
16: return \( d \)

Clearly, the main question is how to determine the set of pairs \( \langle w, d(v, w) \rangle \) in each label \( L_f(v) \) and the set of pairs \( \langle u, d(u, v) \rangle \) in each label \( L_b(v) \) for each \( v \in V \), such that \( L \) has the cover property and such that \( |L_f(v)| \) and \( |L_b(v)| \) are not too large. If the labels contain large sets, not only the running time of Algorithm 6.9 will get higher, but also the storage for all labels will be high. For large graphs, this will surely lead to a high memory consumption.

Definition 6.22 The forward (backward) label size of \( v \) is \( |L_f(v)| \) (|\( L_b(v) |)).
Definition 6.23 Let $L$ be a directed node-labeling on a digraph $\mathcal{G} = (V, A)$. The size of $L$ is defined as $|L| = \sum_{v \in V} (|L_f(v)| + |L_b(v)|)$, while the average size is defined as $|L|/2|V|$. 

Definition 6.24 Let $\mathcal{G} = (V, A)$ be a digraph. The Minimum Label Problem consists of determining a directed node-labeling $L$ that has the cover property such that the size of $L$ is minimal among all such directed node-labelings.

Cohen et al. [33] showed that the Minimum Label Problem can be posed as a Minimum Set Cover Problem (MSCP), which is known to be NP-complete [117]. They use the greedy heuristic for solving the MSCP of Chvátal [32], in order to find a directed node-labeling $L$ that has the cover property, such that the size of $L$ is larger than the size of the solution of the Minimum Label Problem by at most an $O(\log |V|)$ factor. Their algorithm is outlined in Algorithm 6.10.

Algorithm 6.10: Greedy Node Labeling($\mathcal{G}, \mathcal{P}$)

Require: $\mathcal{P}$ is the collection of all canonical shortest paths in $\mathcal{G}$.
1. $Q \leftarrow \{v|v \in V\}$
2. foreach $v \in V$ do
3. \quad $L_f(v) \leftarrow \{(v,0)\}$
4. \quad $L_b(v) \leftarrow \{(v,0)\}$
5. \quad $\mathcal{P}_v \leftarrow \{P|v \in P, P \in \mathcal{P}\}$
6. while $Q \neq \emptyset$ and $\mathcal{P} \neq \emptyset$ do
7. \quad foreach $v \in Q$ do
8. \quad \quad $R_-v \leftarrow \{s|P_{sv} \in \mathcal{P}_v, s \neq v\}$
9. \quad \quad $R_+v \leftarrow \{t|P_{vt} \in \mathcal{P}_v, t \neq v\}$
10. \quad \quad Select a node $v \leftarrow \arg \max_{v \in Q} |\mathcal{P}_v|/\left(|R_-v| + |R_+v|\right)$
11. \quad \quad foreach $w \in R_-v$ do
12. \quad \quad \quad $L_f(w) = L_f(w) \cup \{(v,d(w,v))\}$
13. \quad \quad foreach $w \in R_+v$ do
14. \quad \quad \quad $L_b(w) = L_b(w) \cup \{(v,d(v,w))\}$
15. \quad \quad $Q \leftarrow Q - \{v\}$
16. \quad \quad $\mathcal{P} \leftarrow \mathcal{P} - \mathcal{P}_v$
17. \quad \quad foreach $w \in Q$ do
18. \quad \quad \quad $\mathcal{P}_w \leftarrow \mathcal{P}_w - \mathcal{P}_v$

Determining all canonical shortest paths (see Definition 6.6) in the graph can be done in $O(|V|^3)$ time (see Appendix C). The initialisation of the sets $\mathcal{P}_v$ (line 5) takes $O(|V|^4)$ time. The “while-loop” (line 6) is iterated $O(|V|)$ times and each iteration can be executed in $O(|V|^3)$ time. Here, we assume that the sets $\mathcal{P}_v$ are ordered in some way, such that the update of $\mathcal{P}_w$ in line 18 can be done in $O(|V|^2)$ time. Hence, the execution of Algorithm 6.10 takes $O(|V|^4)$ time.

We follow [2] in introducing the concept of a hierarchical and a canonical labeling.
6.4 Contraction Hierarchies

Definition 6.25 Let $L$ be a directed node-labeling on a digraph $G = (V, A)$. We write $v \preceq w$ if $w \in L_f(v) \cup L_b(v)$. The labeling $L$ is a hierarchical labeling if ‘$\preceq$’ defines a partial ordering on the nodes (see Definition 3.22).

Proposition 6.26 The labeling $L$ that results after running Algorithm 6.10, is hierarchical.

Proof: We prove this by contradiction. Suppose the labeling $L$ is not hierarchical. Then, there exists a triple of nodes $u, v, w$ such that $u \in L_f(v) \cup L_b(v)$, $v \in L_f(w) \cup L_b(w)$ and $w \in L_f(u) \cup L_b(u)$. According to Algorithm 6.10, node $u$ can only be added to the label of another node during the iteration where node $u$ is selected in line 10. At the beginning of that iteration, $P_u$ contains all currently uncovered paths that contain node $u$. At the end of the iteration, all paths that contain node $u$ are covered since $P$ contains only canonical shortest paths. Hence, the forward and backward label of node $u$ cannot be changed in any later iteration. Hence, if node $u$ is added to a label of node $v$, that means node $v$ is selected after node $u$. If node $v$ is added to a label of node $w$, node $w$ is selected after node $u$ as well. It is impossible that node $w$ will be added to a label of node $u$, since there is no uncovered path that contains node $u$ at that moment. This contradicts our assumption. \hfill \qed

Definition 6.27 Let $L$ be a labeling on a digraph $G = (V, A)$ and let ‘$<$’ be a linear ordering on the nodes. $L$ is a hierarchical labeling with respect to ‘$<$’ if $L$ contains only the following label entries: For every canonical shortest path $P_{st}$, $\langle v, d(s, v) \rangle \in L_f(s)$ and $\langle v, d(v, t) \rangle \in L_b(t)$, where $v = \max \{ w | w \in P_{st} \}$ (hence, $v$ is the maximal node in $P_{st}$ with respect to ‘$<$’).


Proof: Let ‘$<$’ be the linear ordering on which the canonical labeling $L$ is based. Since $L$ is canonical with respect to ‘$<$’, for every canonical shortest path $P_{st}$, only the highest (with respect to ‘$<$’)) node $v \in P_{st}$ is element of $L_f(s)$ and $L_b(t)$. Thus, $s \leq v$ and $t \leq v$. Hence, for any $w \in L_f(v) \cup L_b(v)$ we have $v \leq w$. Since ‘$<$’ is a linear ordering, the labeling $L$ is a hierarchical labeling. \hfill \qed

Proposition 6.29 Let $L$ be a hierarchical labeling. The set of nodes on any shortest path has a unique maximum element with respect to the partial order implied by $L$.

Proof: This is proved by induction on the number of nodes on the path. If the path contains only one node, this node is clearly a unique maximum. Now, consider a path $P = (v_1, v_2, v_3, \ldots, v_{k-1}, v_k)$ with $k > 1$. By the induction hypothesis, the subpath $(v_1, \ldots, v_{k-1})$ has a unique maximum node $v_i$. Consider the subpath $(v_i, v_{i+1}, \ldots, v_k)$. The internal nodes of this subpath are not in $L_f(v_i)$ by the choice of $v_i$. Therefore, either $v_i \in L_b(v_k)$ ($v_i$ is the unique maximum node on $P$), or $v_k \in L_f(v_i)$ ($v_k$ is the unique maximum node of $P$). Hence, in both cases a unique maximum node exists. \hfill \qed
A partial ordering ‘≤’, related to a hierarchical labeling $L$, can clearly be embedded in one or more linear orderings. From Proposition 6.29, it follows that all these linear orderings have the same canonical labeling $L'$.

**Proposition 6.30** Let $L$ be a hierarchical labeling and ‘<’ a linear ordering on the nodes, such that $L$ respects ‘<’. Let $L'$ be the canonical labeling for ‘<’. Then, $L'$ is contained in $L$. That is, for every node: $L'_f(v) \subseteq L_f(v)$ and $L'_{b}(v) \subseteq L_{b}(v)$.

**Proof:** The labeling $L'$ contains only the following label entries: $⟨v, d(s, v)⟩ \in L'_f(s)$ and $⟨v, d(v, t)⟩ \in L'_b(t)$, where $v$ is the highest node on the canonical shortest path $P_{st}$. We only have to prove that $⟨v, d(s, v)⟩ \in L_f(s)$ and that $⟨v, d(v, t)⟩ \in L_b(t)$ as well. Consider the shortest path $P_{sv}$. Since the shortest paths are canonical, we have $P_{sv} \subseteq P_{st}$ and thus $v$ is the highest node on $P_{sv}$. Since $L$ is a hierarchical labeling, the only node on $P_{sv}$ that is in $L_{b}(v)$, is node $v$ itself. Since $L$ satisfies the cover property, it follows that $⟨v, d(s, v)⟩ \in L_f(s)$. A similar argument shows that $⟨v, d(v, t)⟩ \in L_b(t)$.

Given an ordering ‘<’ on the nodes, the corresponding canonical labeling has the smallest possible size among all hierarchical labelings that respect ‘<’. Before discussing several ways to choose an appropriate node ordering, we first describe a way to extract a canonical labeling $L'$ from a hierarchical labeling $L$. This can easily be done by pruning the labels in $L$. Consider a node $w$ in $L_f(v)$. Node $w$ can be removed from $L_f(v)$ if it is not the highest node on the canonical shortest path $P_{uw}$. Suppose the highest node on $P_{uw}$ is $u \neq w$. Then, $u$ must be in both $L_f(v)$ and $L_b(w)$. Thus, the labels $L_f(v)$ and $L_b(v)$ should be inspected to see if there exists such a node $u$. If so, the entry with node $w$ is removed from $L_f(v)$. The backward labels can be pruned similarly. Pruning the node labels this way is originally described in [1]. Let $M$ be the maximum label size, that is $M = \max_{v \in V}(|L_f(v)|, |L_b(v)|)$. Pruning one entry of a label takes $O(M)$ time. Pruning all label entries takes $O(|L|M) = O(|V|M^2)$ time.

Abraham et al. [1] observed that the preprocessing of Contraction Hierarchies (Algorithm 6.7) can be used in order to find rather small labels in a reasonable amount of time. For the graph $\mathcal{G} = (V, A)$ resulting from this preprocessing, we know from Proposition 6.12 that for every pair of nodes $s,t \in V$, there exists a shortest $s$-$t$ path $P = (s = u_0, (u_0, u_1), \ldots, u_p, \ldots, (u_{k-1}, u_k), u_k = t)$ with $u_i < u_j$ for $0 \leq i < j \leq p$, and $u_i > u_j$ for $p \leq i < j \leq k$. The forward label $L_f(v)$ of node $v$ is determined by growing a shortest path tree rooted at $v$ in $\mathcal{G}_t = (V, A_t)$, and the backward label $L_b(v)$ of node $v$ is determined by growing a shortest path tree rooted at $v$ in the reverse graph of $\mathcal{G}_t = (V, A_t)$. In this way it is ensured that $L$ has the cover property. It should be noted that these labels are not strict: It might happen that a node $w$ is included in $L_f(v)$ for which the distance in the shortest path tree in $\mathcal{G}_t = (V, A_t)$ is higher than $d(v, w)$ in $\mathcal{G} = (V, A)$. A same argument holds for the backward label $L_b(v)$. Abraham et al. therefore call these labels super labels. Although the labels are not strict, they actually have the cover property and can be used in Algorithm 6.9. Furthermore, these labels can be pruned by inspecting...
them one by one in a top-down manner. Hence, start with the node that is decided to have the highest level during the Contraction Hierarchies preprocessing and work all the way down to the node that is bypassed first. An average (super) label size of 500 is reported for road maps with a continental size, such as Western Europe with $|V| = 18$ million and $|A| = 22.5$ million. After pruning, the initial size of the super label is reduced by about 80%, resulting in an average label size of 85 for the Western Europe road map.

However, the node ordering that is based on Contraction Hierarchies, does not necessarily results in the smallest possible (canonical) labeling. Abraham et al. [2] introduced a top-down approach. Where during the Contraction Hierarchies preprocessing nodes are bypassed from the least important to the most important node, the top-down approach works the other way around. The Top-Down algorithm (Algorithm 6.11) is quite similar to Algorithm 6.10. Commonly, it starts with a collection of all canonical shortest paths. Abraham et al. introduce two versions, based on a different selection rule for the next node (line 10). At the end of the algorithm, a partial ordering `≺` between the nodes is determined: $v \prec w$ if and only if $\pi_v > \pi_w$.

- The covering criterion picks node $v \leftarrow \arg\max\{|P_v| : v \in V\}$.
- The weighted covering criterion picks node $v \leftarrow \arg\max\{|P_v|/(|R_v^-| + |R_v^+|)|}$.

**Algorithm 6.11: Top-Down**

```plaintext
Require: $P$ contains all uncovered canonical shortest paths in $G$.
1: $Q \leftarrow \{v | v \in V\}$
2: foreach $v \in V$ do
3: $P_v \leftarrow \{P | v \in P, P \in P\}$
4: $\pi_v \leftarrow -1$
5: $k \leftarrow 1$
6: while $Q \neq \emptyset$ and $P \neq \emptyset$ do
7: foreach $v \in Q$ do
8: $R_v^- \leftarrow \{s | P_{sv} \in P_v, s \neq v\}$
9: $R_v^+ \leftarrow \{t | P_{vt} \in P_v, t \neq v\}$
10: select $v$ according to a certain criterion based on $P_v$
11: $\pi_v \leftarrow k$
12: $k \leftarrow k + 1$
13: $Q \leftarrow Q - \{v\}$
14: $P \leftarrow P - P_v$
15: foreach $w \in Q$ do
16: $P_w \leftarrow P_w - P_v$
17: Return $\pi$
```

Abraham et al. describe a data structure that can be used to efficiently implement the sets $P_v$ ($v \in V$) and the sizes of $|R_v^-|$ and $|R_v^+|$, based on storing $P$ as a set of $|V|$ shortest path trees. Using this data structure, both versions of Algorithm 6.10 run
in \( \Theta(|V|(|V| \log |V| + |A|)) \) time. Although the top-down approach results in better orderings than contraction hierarchies, for large graphs the running time is still too large for practical applicability. Here, a better ordering means that the size of the corresponding canonical labeling is smaller.

In [1], a strategy is found that can be seen as a combination of both a bottom-up (contraction hierarchies) and a top-down approach. First, the input graph is contracted (by running Algorithm 6.7). But instead of running this algorithm for all \(|V|\) iterations, it is stopped as soon as the resulting graph has a reasonable small number of nodes (up to 10,000 nodes for example). On the resulting graph, the top-down approach is continued in order to find a linear ordering on all nodes. Compared to plain contraction hierarchies, only the ordering of the most important nodes can be improved.

In [2] another combination is proposed, called \textit{range optimization}. The idea is that the ordering among lower ranked nodes can be improved as well. As input, an initial ordering ‘\(<\’ is given as well as two parameters \(i\) and \(j\) (\(i < j\)). The ordering ‘\(<\’ defines the linear sequence \(v_1, v_2, \ldots, v_{|V|}\). The \textit{Range Optimization} procedure (see Algorithm 6.12) might rearrange the nodes \(v_{i+1}, v_{i+2}, \ldots, v_j\) in this sequence. First, the nodes \(v_1, v_2, \ldots, v_i\) are bypassed (lines 1-3). Next, the shortest path trees rooted at \(v_{i+1}, v_{i+2}, \ldots, v_j\) are determined in order to find all paths between a node \(v_k\) and \(v_n\) (\(i < k, n \leq j\)) that contain no nodes higher than \(v_j\). Hence, \(P\) contains only paths that are not covered by a node higher than \(v_j\). Since we only search for such paths, we can use an advanced stop criterium in \textsc{Dijkstra} (line 6): Stop as soon either \(Q = \emptyset\) or \(Q\) contains only nodes with a predecessor higher than \(v_j\). By adding an additional flag to each node, stating whether or not it contains a predecessor higher than \(v_j\) in the shortest path tree, and by bookkeeping the number of such nodes in \(Q\), this extra stop condition can be checked in \(O(1)\) time. Taking only into account the shortest path that contains no higher nodes than the input range, speeds up both the calculation of the shortest path trees and the execution of the \textit{Top-Down} procedure.

To allow a node to move between arbitrary positions, an iterative execution of the \textit{Range Optimization} procedure is proposed. The interval \([1, |V|]\) is covered by \(n\) overlapping intervals \([i_k, j_k]\), with \(i_1 = 0, j_n = |V|, i_k < j_k\) and \(i_{k+1} \leq j_k\). One can start with an arbitrary node ordering. Commonly, this will be the one used for the Contraction Hierarchies preprocessing. Iteratively, for each \(k = 1, \ldots, n\) the corresponding subrange \([i_k, j_k]\) is rearranged. Abraham et al. reports a label size of 69 for the earlier mentioned graph of Western Europe. This is an improvement of about 19\%, compared to the labeling that was based on the node ordering used with Contraction Hierarchies.

In [2], another algorithm (Algorithm 6.13) to determine a canonical labeling based on a node ordering ‘\(<\’, is proposed. This algorithm closely resembles the ideas behind a recent algorithm, called \textsc{Snowball} (see Appendix C). First, the Contraction Hierarchies preprocessing (Algorithm 6.7) is executed. Subsequently the labels are recursively determined in a top-down approach one by one. After the execution of Algorithm 6.13, the labels can be pruned in the same way as described above.
6.4 Contraction Hierarchies

Algorithm 6.12: RANGE OPTIMIZATION($\mathcal{G},<,i,j$)

1: for $k = 1$ to $i$ do
2:   Bypass node $v_i$ in $\mathcal{G}$ by constructing $\overline{\mathcal{G}}$ (see Definition 6.8)
3:   $\mathcal{G} \leftarrow \overline{\mathcal{G}}$
4:   $\mathcal{P} \leftarrow \emptyset$
5: for $k = i + 1$ to $j$ do
6:   DIJKSTRA($\mathcal{G}, v_k$)
7:   for $n = i + 1$ to $j$ do
8:      if $w \in [v_{i+1}, v_{i+2}, \ldots v_j]$ for all $w \in P_{v_k,v_n}$ then
9:         $\mathcal{P} \leftarrow \mathcal{P} \cup P_{v_k,v_n}$
10:      end if
11:   end for
12:   TOP-DOWN($\mathcal{G}, \mathcal{P}$) (this results in a partial ordering `$\prec$')
13:   rearrange $v_{i+1}, v_{i+2}, \ldots, v_j$ such that `$\prec$' is obeyed
14: return the new node ordering

Algorithm 6.13: RECURSIVE LABELING($\mathcal{G},<$)

1: CONTRACTION HIERARCHIES($\mathcal{G},<$)
2: for $k = |V|$ downto 1 do
3:    $L_f(v_k) \leftarrow \{\langle v_k, 0 \rangle \}$
4:    $L_b(v_k) \leftarrow \{\langle v_k, 0 \rangle \}$
5: foreach $(v_k, v_i) \in A_i^{-1}$ do
6:   foreach $(u, d_{u,v_i}) \in L_f(v_i)$ do
7:      if $(u, d_{u,v_i}) \in L_f(v_k)$ then
8:         $d_{v_k,u} \leftarrow \min\{d_{v_k,u}, c_{v_k,v_i} + d_{v_i,u}\}$
9:      else
10:         $L_f(v_k) \leftarrow L_f(v_k) \cup \{u, c_{v_k,v_i} + d_{v_i,u}\}$
11:    end if
12:   end foreach
13: end foreach
14: foreach $(v_i, v_k) \in A_i$ do
15:   foreach $(u, d_{u,v_k}) \in L_b(v_k)$ do
16:      if $(u, d_{u,v_k}) \in L_b(v_k)$ then
17:         $d_{u,v_k} \leftarrow \min\{d_{u,v_k}, d_{u,v_i} + c_{v_i,v_k}\}$
18:      else
19:         $L_b(v_k) \leftarrow L_b(v_k) \cup \{u, d_{u,v_i} + c_{v_i,v_k}\}$
20:    end if
21: end foreach
6.5 Hybrid graph preprocessing

So far, we distinguished between static graph preprocessing and dynamic graph preprocessing, based on the question whether or not the preprocessing depends on the arc costs. Static graph preprocessing is more robust. A change in the arc costs has no influence on the static preprocessing. Since dynamic preprocessing explicitly uses the arc costs as input, even a minor change in the arc costs will lead to some additional work to be done. However, the amount of work can vary from recalculating the distances from and to a landmark \( L \) in an otherwise unchanged set of landmarks \( L \) (Section 6.3.3), or to restart the preprocessing from scratch, for instance the calculation of reaches (Section 6.3.2).

So far, we used a single arc cost function \( C : A \to \mathbb{R} \) that assigns a cost to each arc in \( A \). In this paragraph, we focus on the situation where a set \( C \) of arc cost functions is given. For example, \( C \) might contain a specific arc cost function for a specific season, day of the week and/or time of the day.\(^2\) Each time a shortest path has to be calculated, an arc cost function \( C \in C \) is chosen, which best describes the situation (season, day, time). Hence, it is necessary to either store multiple arc costs with each arc, or to quickly update the cost of each arc. Furthermore, new cost functions might be added to the set \( C \) as well. Such a new cost function can be used to model actual traffic situations, like unforeseen congestions, etc. Nowadays, it is possible to get actual traffic speeds for the most important parts of the road map, each minute.\(^3\) Clearly, real-time arc costs cannot be stored in advance. Hence, it should be possible to quickly update the arc costs.

In this paragraph, we describe how we still can use preprocessing in such a case. We still consider the arc costs to be constant during the execution of a shortest path algorithm. One could also model time dependent arc costs. In that case, the arc cost \( c_{vw}(t) \) depends on the arrival time \( t \) on node \( v \). The time-dependent shortest path problem TDSPP was first described by Cooke and Halsey [34]. In Appendix D, we will describe the TDSPP in more detail.

Delling et al. [49] introduced the term customizable route planning. They distinguish between two feature types of an arc in a road map, topology and metric. The topology consists of static properties of the arcs, like their physical length, road class, speed limits, etc. The metric is the actual cost of traversing the arc. We prefer the term arc costs instead of metric, since a metric assumes symmetry (see Definition 6.7). Several arc cost functions can be combined with one topology. Preprocessing can be done in a hybrid way that combines static and dynamic methods as follows. First, an arc cost independent (static) preprocessing is done, based on the graph topology only. It is no problem that this step is relatively slow, since it is executed only once. This static preprocessing might lead to a fair amount of auxiliary data. The second step is a dynamic preprocessing step based on a specific arc cost function. This preprocessing is executed once for each arc cost function considered. Therefore, it must be fast and only little auxiliary data may be created.

\(^2\)For instance, TomTom supplies so-called speed profiles for different days and times.
\(^3\)TomTom provides such a product, called HD Flow.
The amount of auxiliary data resulting from a dynamic preprocessing might be independent of the arc cost function that is actually used. This is for instance the case with the calculation of landmarks and reaches. Other dynamic preprocessing methods, especially the ones where nodes are bypassed, result in an amount of auxiliary data (shortcuts) that depends on the actual arc cost function used. In case of Contraction Hierarchies for instance, it might be possible to use the same node ordering for different arc cost functions. However, the number of shortcuts that are needed for two different cost functions, can vary a lot. In [13], it is observed that dynamic preprocessing (like landmarks, reaches and Contraction Hierarchies) leads to fast shortest path algorithms if the arc cost function that is considered, is hierarchical (for instance, if the arc costs represent travel times). For less pronounced arc costs (like distances), the resulting shortest path algorithms are not that fast.

Delling et al. [49] exploit the concept of graph partitioning during the static preprocessing step. They use the so-called punch algorithm [50]. Given an input parameter $n$, this algorithm partitions the node set of $G = (V, A)$ in cells $V_1, V_2, \ldots, V_K$ such that $|V_i| \leq n$, and the subgraph $G_i \subset G$ induced by $V_i$ is weakly connected (for $1 \leq i \leq K$). Furthermore, the number of boundary arcs is minimized. An arc $(v, w)$ is a boundary arc if $v \in V_k$, $w \in V_l$ and $k \neq l$. If $(v, w)$ is a boundary arc, the nodes $v$ and $w$ are called boundary nodes. During the dynamic preprocessing step a graph $G_0 = (V_0, A_0)$ is constructed as follows: $V_0 \subset V$ contains all boundary nodes, $A_0$ contains all boundary arcs. For each cell $V_i$ ($1 \leq i \leq K$), the subgraph $G_i$ of $G$ induced by $V_i$ is considered. A shortest $v$-$w$ path in $G_i$ is determined for any pair of boundary nodes in $V_i$. If there exists such a shortest path, an arc $(v, w)$ is added to $A_0$, where $c_{vw}$ equals the length of the shortest path found.

Clearly, this way of graph partitioning is not equal to the one described in Section 6.2.3. In that section, first a set of core nodes $V_0$ was determined and no shortcuts were added to the core graph $G_0$ during the partitioning. However, the resulting shortest path algorithm is comparable. If the shortest path from node $s \in V_k$ to node $t \in V_l$ has to be determined, the graph that is considered is the union of $G_0$, $G_k$ and $G_l$. In [49], a bidirectional version of Dijkstra is used for solving the shortest path problem on this graph. Hence, the running time of the shortest path algorithm highly depends on $|V_0|$ and $|A_0|$. Therefore, we discuss the partitioning problem in the next section and we will briefly describe how the punch algorithm tries to solve this problem.

### 6.5.1 The graph partitioning problem

First, we formally describe the graph partitioning problem.

**Definition 6.31** Let $G = (V, E)$ be an undirected graph and $P = (V_1, V_2, \ldots, V_K)$ a partitioning of $V$. An edge \{v, w\} is called a cut-edge (or boundary edge) if $v \in V_k$, $w \in V_l$ and $k \neq l$.

**Definition 6.32** Let $G = (V, E)$ be an undirected graph and $P$ a partitioning of $V$. The cut-edge set $\delta(P)$ consists of all cut-edges with respect to $P$. If $c(e)$ denotes the
cost of edge $e \in E$, the cost of $P$ is defined as:
\[
c(P) := \sum_{e \in \delta(P)} c(e).
\]

**Definition 6.33** Let $G = (V, E)$ be an undirected graph and $n$ a parameter. The graph partitioning problem is to find a partitioning $P = (V^1, V^2, \ldots, V^K)$ of $V$, such that $c(P)$ is minimized and $|V^i| \leq n$, for $1 \leq i \leq K$.

The graph partitioning problem is known to be NP-hard [77]. The PUNCH algorithm solves the problem heuristically in two phases. During the first phase, called filtering, relatively dense areas in the graph are contracted.

**Definition 6.34** Let $G = (V, E)$ be an undirected graph where each node $v \in V$ has a ‘size’ $s(v) \in \mathbb{N}$. To contract two nodes $v$ and $w$, these two nodes are replaced by a new node $x$ with $s(x) \leftarrow s(v) + s(w)$. Moreover, for each edge $\{u, v\}$ or $\{u, w\}$ (with $u \notin \{v, w\}$), a new edge $\{x, u\}$ is created with the same weight as the original one.

Contracting a set of nodes consists of repeatedly contracting pairs of nodes in the set in any order, until a single node remains. Note, that the contraction of nodes $v$ and $w$ might result in a graph with multiple edges. If the original graph $G$ is simple it is easy to keep the graph simple after the contraction of nodes $v$ and $w$. If both $\{v, u\}$ and $\{w, u\}$ exist, we add only one new edge $\{x, u\}$ with $c_{xu} \leftarrow c_{vu} + c_{wu}$.

The filtering phase consists of two stages:

- Identifying tiny cuts;
- Identifying natural cuts.

We will discuss both stages in the next two sections.

### 6.5.1.1 Identifying tiny cuts

Identifying tiny cuts is done in three consecutive steps:

- Identifying biconnected parts of the graph;
- Identifying nodes with degree 2;
- Identifying triconnected parts of the graph.

We take the underlying undirected graph $G$ of our road map as input. We assume that $G$ is connected and simple. The procedure is initialized by setting $c_{vw} \leftarrow 1$ for each $\{v, w\} \in E$ and $s(v) \leftarrow 1$ for each $v \in V$. After identifying all biconnected parts of the graph, each biconnected part that contains less than $n$ nodes is contracted to a single node $x$. Hence, the size of that single node $x$ represents the number of the original nodes in the biconnected part. Node $x$ is now connected to just one other
node $y$ ($y$ is in another biconnected part of the graph). If $s(x) + s(y) \leq n$ nodes $x$ and $y$ are contracted in turn.

Next, we determine all nodes with degree two. Any degree two node lies on a unique path $P$ whose interior nodes have degree two. If the sum of the sizes of all interior nodes of $P$ are less than $n$, these nodes are contracted.

These first two steps are highly comparable with the Static preprocessing procedure (Algorithm 6.3). At the end of that procedure, a partitioning of $\mathcal{V} = (\mathcal{V}^0, \mathcal{V}^1, \mathcal{V}^2, \ldots, \mathcal{V}^K)$ is found. As stated in Table 6.2, the largest non-core component in our test graph only has 2,127 nodes. Commonly, the punch procedure takes a parameter $n$ much higher than 2,127 as input. The first two steps of the punch algorithm can be emulated by running the Static preprocessing procedure followed by the contraction of each non-core cell $\mathcal{V}^i$ ($1 \leq i \leq K$) to a new node $v_i$. Node $v_i$ has either one or two adjacent core nodes in $\mathcal{V}^0$. Node $v_i$ is contracted with that neighbors.

The third step requires some more work. We briefly describe this procedure.

**Definition 6.35** A connected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is triconnected if and only if $\mathcal{G}$ remains connected after the deletion of two nodes (and their incident edges).

Determining the triconnected components of a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ can be done in $O(|\mathcal{V}| + |\mathcal{E}|)$ time using an algorithm by Hopcroft and Tarjan [107]. Their elegant article contains some minor yet crucial errors that were corrected by Gutwenger and Mutzel [100]. Recently, Tsin [191] provided another algorithm with the same time bounds, but which needs considerably less operations to complete. Pritchard and Thurimella [161] presented an algorithm that finds all pairs of edges whose deletion result in an unconnected graph. Their algorithm runs in $O(D + \delta / \log |\mathcal{V}|)$ time, where $\delta$ is the maximum degree of all nodes and $D$ is the diameter of $\mathcal{G}$ (see Definition 2.42).

All triconnected components that contain at most $n$ nodes, are contracted.

6.5.1.2 Identifying natural cuts

After identifying the tiny cuts of the graph, a second stage is executed in order to find the ‘natural cuts’. Road maps are not uniform, there are dense regions close to natural cuts, like bridges, mountain passes, ferries, and country borders. This stage is the innovative part of the punch algorithm and explains its name: ‘PUNCH’ is the acronym of ‘Partitioning Using Natural Cuts Heuristics’. The way possible natural cuts are identified is outlined in Algorithm 6.14. The input for this algorithm is an undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$, an initial node size $s(v), v \in \mathcal{V}$ and edge weights $c(e), e \in E$, together with three parameters $\alpha$ ($0 < \alpha \leq 1$), $f > 1$ and $n$. We shortly write $s(T), T \subseteq \mathcal{V}$, to denote $\sum_{v \in T} s(v)$.

The idea is to assign each node $v \in \mathcal{V}$ to a so-called core-region. The procedure iterates as long as there are nodes left unassigned to any core (that is, as long as
During each iteration, an unassigned node $v$ is randomly selected. A Breadth-First Search (BFS) procedure is started in order to grow a tree $T$ rooted at $v$. This procedure is stopped as soon as $s(T)$ reaches $\alpha n$. Furthermore, a core region $C$ of $v$ is determined as the union of all nodes added to $T$, before its size reached $\alpha n/f$. The neighbors of $T$ that are in $V \setminus T$ (denoted as $\delta(T)$) are called the ‘ring’ $R$ of $v$. The minimum (with respect to $c$) cut between $R$ and $C$ is determined. The cut-edges are added to $E'$. At the end of Algorithm 6.14, a set of ‘natural cut edges’ is determined. Figure 6.12 depicts the core region and the ring of a node $v$, where $n = 32768$, $\alpha = 1$ and $f = 10$.

**Figure 6.12:** The core-region (orange) and ring (nodes on the solid black outer line) of a node $v$ (black dot), created by running a BFS.

In order to determine the minimum cut between two sets of nodes $R$ and $C$, the nodes in $R$ are temporarily contracted to one node $s$ and the nodes in $C$ are temporarily contracted to one node $t$. Next, any $s$-$t$ cut algorithm can be used. Many $s$-$t$ cut algorithms are based on the famous Max-Flow-Min-Cut theorem by Ford and Fulkerson [71]. These algorithms start with an empty flow $x$. Iteratively, extra flow is pushed from $s$ to $t$ along augmenting paths, until the maximum flow is found. An augmenting path is a path that contains only non-saturated edges. Commonly a so-called residual graph $G_x$ is used that has the same nodes and edges as the original graph $G$, but the capacity of an edge in $G_x$ reflects the residual capacity of the same edge in $G$ given the amount of flow already on that edge. If an augmenting path is found in $G_x$ the flow $x$ is augmented by pushing the maximum possible flow along that path. Hence, at least one of the edges of the path becomes saturated. The residual capacities in $G_x$ are updated. The maximum flow is found as soon as no augmenting $s$-$t$ path exists in the residual graph. Well-known algorithms are the one of Edmonds and Karp [63] that use breadth-first search from $s$ while searching for an augmenting $s$-$t$ path. Their algorithm runs in $O(|V||E|^2)$ time. A similar algorithm of Dinic [60] uses some additional techniques and has a complexity of $O(|V|^2|E|)$. Delling et al. [50] suggest the push-relabel method of Goldberg and Tarjan [95]. That algorithm runs in $O(|V||E| \log(|V|^2/|E|))$ time. In the push-relabel method, a flow is not always valid. it is allowed that some so-called ‘active’ nodes have a
flow-excess. For each node \( v \) an estimate on the distance to \( t \) along non-saturated edges is maintained. This estimate is always a lower bound to the actual distance. It is attempted to ‘push’ excess flow towards nodes with smaller estimated distance to \( t \). This push operation is applied to active nodes, either guided (starting with an active node that has the largest estimated distance to \( t \)), or using a first-in, first-out approach. The estimated distances increases as edges become saturated by push operations. This is the ‘re-label’ part of the method. Undeliverable flows are eventually drained back to \( s \).

An algorithm of Boykov and Kolmogorov [22] turns out to be fast in practical evaluations, although the worst-case complexity is worse than any of the abovementioned \( s-t \) cut algorithms. The algorithm works in a bidirectional way. The search for an \( s-t \) path is done by growing two search trees: one rooted at \( s \) and one rooted from \( t \). Furthermore, these trees are never rebuild from scratch during the algorithm. Each node \( v \in \mathcal{V} \) belongs to \( S \) (the tree rooted as \( s \)) or to \( T \) (the tree rooted at \( t \)), or is still ‘free’:

\[
S \subseteq \mathcal{V}, \quad s \in S, \quad T \subseteq \mathcal{V}, \quad t \in T, \quad S \cap T = \emptyset.
\]

The leaves of the trees \( S \) and \( T \) are called ‘active’. All other nodes in \( S \) and \( T \) are called ‘passive’. Active nodes allow a tree to grow, by acquiring new children (along non-saturated edges) from a set of free nodes. As soon as an active node tries to acquire a child node that is part of the other tree, an augmenting path is found. The algorithm iteratively runs the following steps:

- Grow search trees \( S \) and \( T \) until an \( s-t \) path is found (or no free nodes exists, in which case the algorithm terminates).
- Augment the flow along the \( s-t \) path found. Break the search trees into forests.
- Restore the search trees \( S \) and \( T \).

A set of active nodes \( Q \) is maintained during the algorithm. An active node \( v \in Q \) is selected. The active node \( v \) belongs either to \( S \) or to \( T \). Via non-saturated edges it is tried to acquire new children for \( v \). If a possible child is part of the other search tree an augmenting path is found. Otherwise, node \( v \) becomes passive and is removed from \( Q \). If an augmenting path is found, the largest possible flow is pushed along that path. At least one of the edges in the path becomes saturated. Thus, some nodes in \( S \) or \( T \) becomes ‘orphans’. The edge that links an orphan to its parent is not valid anymore (since it is saturated). Each orphan forms the root of a new tree. Instead of only two search trees, we now have a forest of more than two trees. Note that this forest always contains a tree rooted at \( s \) and one rooted at \( t \). The orphans have to be adopted to a new parent that is in the same set \( S \) or \( T \) as the orphan already was. Of course, such a parent should be connected to the orphan via a non-saturated edge. If no such parent can be found, the nodes in the tree rooted at the orphan are all set to be free. Algorithm 6.15 describes this algorithm more formally. The two subroutines \textsc{Augment} and \textsc{RestoreTrees} are stated in Algorithms 6.16 and 6.17. The \textsc{Augment} subprocedure first determines the extra amount of flow \( \Delta \) that can be sent from \( s \) to \( t \). Then, the flow on the edges in the \( s-t \) path is increased by \( \Delta \). If an edge becomes saturated, and both adjacent nodes
belong to the same search tree (either $S$, or $T$), the child node is added to $O$, the set of all orphans. The \textsc{RestoreTrees} subprocedure picks, one by one, an orphan $v$ from $O$. First, it is attempted to find a new parent $u$ for $v$. Such a parent should satisfy the conditions that $(u, v) \in E$ and both $u$ and $v$ belong to the same search tree. Furthermore, the edge $(u, v)$ must be non-saturated and the root of $u$ must be either $s$ or $t$. If no such parent is found, each neighbor $w$ of $v$ are scanned. If $w$ belongs to the same search tree as $v$ and the edge $(v, w)$ is non-saturated, node $w$ is added to the active set $Q$. If $\pi_w = v$ (node $w$ is a child of $v$), node $w$ becomes an orphan itself. Finally, node $v$ is removed from its search tree and hence from $Q$ as well.

After determining a set of natural cut-edges $E' \subseteq E$, all other edges are contracted one by one. Each remaining node represents a fragment of the original graph. By choosing the parameters $\alpha \leq 1$ and $f > 1$ it is ensured that the number of original nodes that belongs to the fragment is at most $n$. Due to the random factor in Algorithm 6.14, it is observed [50] that running the algorithm more than once and adding the cut-edges found in each run together, might lead to a better result. However, if the final cut-edge set is too large, the final graph partition might get worse. Delling et al. therefore recommend to run Algorithm 6.14 only once or twice. In our experiments, we always ran the algorithm twice.

Figure 6.13: After the filtering phase several natural cuts are detected in the input graph (left). All other edges are contracted. Each remaining node of the resulting graph (right) represents a fragment of the input graph.
6.5 Hybrid graph preprocessing

Algorithm 6.14: **IDENTIFYNATURALCUTS**($\mathcal{G} = (\mathcal{V}, \mathcal{E}), s, c, \alpha, f, n$)

1: $\mathcal{E}' \leftarrow \emptyset$
2: $\mathcal{V}' \leftarrow \mathcal{V}$
3: while $\mathcal{V}' \neq \emptyset$ do
4:  Pick randomly a node $v \in \mathcal{V}'$
5:  BFS($v$)
6:  $\mathcal{R} \leftarrow \emptyset$
7:  foreach $\{u, w\} \in \delta(\mathcal{T})$ do
8:     if $u \in \mathcal{T}$ then
9:        $\mathcal{R} \leftarrow \mathcal{R} \cup \{w\}$
10:    else
11:       $\mathcal{R} \leftarrow \mathcal{R} \cup \{u\}$
12:  $\mathcal{G}'$ be the subgraph of $\mathcal{G}$ induced by $\mathcal{T} \cup \mathcal{R}$
13:  $\mathcal{E}'' \leftarrow$ DETERMINEMINCUT($\mathcal{G}', c, \mathcal{C}, \mathcal{R}$)
14:  $\mathcal{E}' \leftarrow \mathcal{E}' \cup \mathcal{E}''$
15:  $\mathcal{V}' \leftarrow \mathcal{V}' - \mathcal{C}$
16: return $\mathcal{E}'$

procedure: BFS($v$)

17: $\mathcal{T} \leftarrow \emptyset$
18: $\mathcal{C} \leftarrow \emptyset$
19: $\mathcal{Q}$.CREATE
20: $\mathcal{Q}$.ENQUEUE($v$)
21: while $\mathcal{Q} \neq \emptyset$ and $s(\mathcal{T}) + s(\mathcal{Q}$.PEEK$) \leq \alpha \cdot n$ do
22:  $w \leftarrow \mathcal{Q}$.DEQUEUE
23:  $\mathcal{T} \leftarrow \mathcal{T} \cup \{w\}$
24:  if $s(\mathcal{C}) + s(w) \leq \alpha n / f$ then
25:    $\mathcal{C} \leftarrow \mathcal{C} \cup \{w\}$
26:  foreach $\{u, w\} \in \mathcal{E}$ do
27:    if $u \notin \mathcal{T}$ and $u \notin \mathcal{Q}$ then
28:        $\mathcal{Q}$.ENQUEUE($u$)
Algorithm 6.15: DetermineMinCut($G = (\mathcal{V}, \mathcal{E}), c, C, R$)

1: $G' \leftarrow G$
2: Contract $C$ to one new node $s$ in $G'$
3: Contract $R$ to one new node $t$ in $G'$
4: $Q \leftarrow \{s, t\}$, $S \leftarrow \{s\}$, $T \leftarrow \{t\}$, $\pi_s \leftarrow \text{NIL}$, $\pi_t \leftarrow \text{NIL}$
5: foreach $\{v, w\} \in E'$ do
6: \hspace{1em} $x_{vw} \leftarrow 0$
7: \hspace{1em} $x_{wv} \leftarrow 0$
8: while $Q \neq \emptyset$ do
9: \hspace{1em} Select a node $v$ from $Q$.
10: \hspace{2em} if $v \in S$ then
11: \hspace{3em} foreach $\{v, w\} \in E$ such that $c_{vw} - x_{vw} > 0$ do
12: \hspace{4em} if $w \in T$ then
13: \hspace{5em} $O \leftarrow \text{Augment}(G', c, x, v, w, S, T)$
14: \hspace{5em} $\text{RestoreTrees}(G', c, x, \pi, S, T, Q, O)$
15: \hspace{4em} break
16: \hspace{3em} else if $w \notin S$ then
17: \hspace{4em} $\pi_w \leftarrow v$, $S \leftarrow S \cup \{w\}$, $Q \leftarrow Q \cup \{w\}$
18: \hspace{2em} else
19: \hspace{3em} foreach $\{v, w\} \in E$ such that $c_{vw} - x_{vw} > 0$ do
20: \hspace{4em} if $w \in S$ then
21: \hspace{5em} $O \leftarrow \text{Augment}(G', c, x, v, w, S, T)$
22: \hspace{5em} $\text{RestoreTrees}(G', c, x, \pi, S, T, Q, O)$
23: \hspace{4em} break
24: \hspace{3em} else if $w \notin T$ then
25: \hspace{4em} $\pi_w \leftarrow v$, $T \leftarrow T \cup \{w\}$, $Q \leftarrow Q \cup \{w\}$
26: \hspace{2em} if no augmenting path was found via $v$ then
27: \hspace{3em} $Q \leftarrow Q - \{v\}$

// Determine cut edges
28: $E'' \leftarrow \emptyset$
29: foreach $\{v, w\} \in E$ do
30: \hspace{1em} $v' \leftarrow \text{ContractedNode}(v)$
31: \hspace{1em} $w' \leftarrow \text{ContractedNode}(w)$
32: \hspace{1em} if $(v' \in S) \neq (w' \in S)$ then
33: \hspace{2em} $E'' \leftarrow E'' \cup \{\{v, w\}\}$
34: return $E''$

procedure: ContractedNode$(v)$

35: $v' \leftarrow v$
36: if $v \in C$ then
37: \hspace{1em} $v' \leftarrow s$
38: else if $v \in R$ then
39: \hspace{1em} $v' \leftarrow t$
40: return $v'$
Algorithm 6.16: Augment($G', c, x, \pi, v, w, S, T$)

1: $\Delta \leftarrow \infty$
2: $P \leftarrow \emptyset$
3: $u \leftarrow v$
4: while $\pi_u \neq \text{NIL}$ do
5:     $\Delta \leftarrow \min(\Delta, c_{\pi_u, u} - x_{\pi_u, u})$
6:     $P \leftarrow P \cup \{(\pi_u, u)\}$
7:     $u \leftarrow \pi_u$
8: u$ \leftarrow w$
9: while $\pi_u \neq \text{NIL}$ do
10:    $\Delta \leftarrow \min(\Delta, c_{u, \pi_u} - x_{u, \pi_u})$
11:    $P \leftarrow P \cup \{(u, \pi_u)\}$
12:    $u \leftarrow \pi_u$
13: $O \leftarrow \emptyset$
14: foreach $(i, j) \in P$ do
15:     $x_{ij} \leftarrow x_{ij} + \Delta$
16:     if $c_{ij} = x_{ij}$ then
17:         if $i \in S$ and $j \in S$ then
18:             $\pi_j \leftarrow \text{NIL}$
19:             $O \leftarrow O \cup \{j\}$
20:         else if $i \in T$ and $j \in T$ then
21:             $\pi_i \leftarrow \text{NIL}$
22:             $O \leftarrow O \cup \{i\}$
23:     return $O$
Algorithm 6.17: RESTORETrees($G, c, x, \pi, S, T, Q, O$)

1: while $O \neq \emptyset$ do
2:     $O \leftarrow O - \{v\}$
3:     if $v \in S$ then
4:         $\pi_v \leftarrow u$
5:     else
6:         foreach $(v, w) \in E$ do
7:             if $w \in S$ then
8:                 if $x_{uv} < c_{uv}$ then
9:                     $Q \leftarrow Q \cup \{w\}$
10:                    if $\pi_w = v$ then
11:                        $\pi_w \leftarrow \text{NIL}$
12:                        $S \leftarrow S - \{v\}$
13:                        $Q \leftarrow Q - \{v\}$
14:                else
15:                    if $\exists \{u, v\} \in E$ with $u \in T, x_{uv} < c_{uv}$ and $\text{Root}(u) = t$ then
16:                        $\pi_v \leftarrow u$
17:                    else
18:                        foreach $(v, w) \in E$ do
19:                            if $w \in T$ then
20:                                if $x_{uv} < c_{uv}$ then
21:                                    $Q \leftarrow Q \cup \{w\}$
22:                                if $\pi_w = v$ then
23:                                    $\pi_w \leftarrow \text{NIL}$
24:                                    $S \leftarrow S - \{w\}$
25:                                    $T \leftarrow T - \{w\}$
26:                                    $Q \leftarrow Q - \{w\}$
27:                            else
28:                                $\pi_v \leftarrow u$
29:                if $\exists \{u, v\} \in E$ with $u \in S, x_{uv} < c_{uv}$ and $\text{Root}(u) = s$ then
30:                        $\pi_v \leftarrow u$
31:         else
32:             $\pi_v \leftarrow \text{NIL}$
33:             $r \leftarrow \text{Root}(\pi_v)$
34:     return $r$

procedure: $\text{Root}(v)$

30: if $\pi_v = \text{NIL}$ then
31:     $r \leftarrow v$
32: else
33:     $r \leftarrow \text{Root}(\pi_v)$
34: return $r$
6.5 Hybrid graph preprocessing

6.5.1.3 Assembling the fragments

The assembling phase takes the fragment graph (see Figure 6.13) from the filtering phase as input. Any partition of this contracted graph corresponds to a partition of the original input graph (the one used as input of the filtering phase). The assembling process computes and stores a score for each edge \( \{v, w\} \) in the fragment graph with \( s(v) + s(w) \leq n \):

\[
\text{score}(\{v, w\}) := \left( \frac{c_{vw}}{\sqrt{s(v)}} + \frac{c_{vw}}{\sqrt{s(w)}} \right). \tag{6.12}
\]

It seems logical to contract nodes that are relatively small but tightly connected. Delling et al. observed that in a road network, one might expect that a region of size \( k \) has about \( O(\sqrt{k}) \) outgoing edges. The edge with maximal score is contracted first\(^4\). After this contraction, the score of each edge adjacent to the contracted node is recomputed. Candidate edges for contraction are stored in a priority queue (see Appendix B). An edge \( \{v, w\} \) for which \( s(v) + s(w) > n \) is simply ignored.

In this way, the graph is assembled in a greedy way. Such greedy heuristics can often be improved by techniques such as:

- Multiple runs
- Local search
- Meta-heuristics

Multiple runs with the same score function will lead to exactly the same result. Therefore, Delling et al. proposed to use a random factor \( 0 \leq r \leq 1 \) that is highly biased towards 1. Each time the score (6.12) is calculated it is multiplied by such a random factor. The result of this multiplication is stored and used in the priority queue. After doing that, it makes sense to re-run the assembling phase, since one might expect different results. The best partitioning found, is returned as the final solution.

Delling et al. introduced three possible types of local search algorithms to improve the quality of a solution that is found using the abovementioned greedy approach. Each of them consists of a number of reoptimization steps. The current partition is viewed as a contracted graph \( \mathcal{H} \). Each node of \( \mathcal{H} \) corresponds to a cell of the partition. There is an edge \( \{\mathcal{V}^i, \mathcal{V}^j\} \) between cells \( \mathcal{V}^i \) and \( \mathcal{V}^j \) in \( \mathcal{H} \) if there is at least one edge \( \{v, w\} \) in the fragment graph \( \mathcal{G} \) (the input graph of the assembling phase) with \( v \in \mathcal{V}^i \) and \( w \in \mathcal{V}^j \). During each reoptimization step an auxiliary graph \( \mathcal{G}' = (\mathcal{V}', \mathcal{E}') \) consisting of a connected subset of nodes of \( \mathcal{H} \) is built. Some of those nodes are expanded (that is, decomposed into the original fragments of \( \mathcal{G} \)), while others remain contracted. The weight of an edge in \( \mathcal{E}' \) is given by the sum of the weights of the corresponding edges in \( \mathcal{G} \). The auxiliary graph \( \mathcal{G}' \) in turn, is assembled in the same greedy way. The solution for \( \mathcal{G}' \) can be used to build a modified solution.

\(^4\)In [50] it is abusively written that an edge with minimal score is selected.
\( H' \). If \( H' \) is better than \( H \), \( H \) is replaced by \( H' \). The three types of local search only differ in the way \( G' \) is constructed. All of them are based on one single edge \( \{V^i, V^j\} \) in \( H \). The first version only considers the expanded versions of the cells \( V^i \) and \( V^j \). The second one also included the contracted neighbors of \( V^i \) and \( V^j \). The last one takes the expanded versions of these neighbors into account. This is visualized in Figure 6.14 [50]. Although the three versions of local search differ in the number of nodes of \( G' \), they are all fully described by \( H \) and the edge \( \{V^i, V^j\} \).

A counter \( \varphi_{ij} \) is assigned to each edge \( \{V^i, V^j\} \) in \( H \). This counter is initialized to zero for each edge. Each time \( \{V^i, V^j\} \) is used in a reoptimization step that did not lead to an improvement, the counter \( \varphi_{ij} \) is increased by one. If an improvement \( H' \) is found, the counters for edges having at least one expanded node in \( G' \) as an endpoint are reset to zero. As soon as the counter \( \varphi_{ij} \) of an edge \( \{V^i, V^j\} \) reaches a certain bound \( \varphi \) the edge is dropped as a candidate for improvement. Higher values of \( \varphi \) may lead to better solutions, but surely will increase the running time of the algorithm. We used the value \( \varphi = 16 \) as suggested by Delling et al.

Local search in general will lead to a local optimum. Several so-called meta-heuristics can be used to escape from local optima. Delling et al. described a version where new solutions are generated that are based on a pool of solutions. We briefly mention their strategy. A maximum number of solutions is stored in memory. Each time a solution \( P \) is found it is added to the pool or dropped. \( P \) is added to the pool if either the pool is not yet full, or the solution is better than at least one of the solutions in the pool. In the latter case one solution in the pool is replaced by \( P \). In order to ensure some diversity among the solutions in the pool, of all solutions that are inferior to \( P \) the one that is most similar to \( P \) is replaced by \( P \). For this purpose, the difference between two solutions \( P' \) and \( P'' \) is defined as the cardinality

\[ |P' - P''| \]
of the symmetric difference:
\[
\{ (V_i, V_j) \mid (V_i, V_j) \in P' \cup P'', (V_i, V_j) \notin P' \cap P'' \}
\]
between their set of cut edges. This replacement strategy originates from [162]. A new solution is generated from two parents \( P' \) and \( P'' \). These parents might both be taken randomly from the pool or one of them is taken from the pool, while the other is generated from scratch. For the new solution, which can be seen as a child of \( P' \) and \( P'' \) the following steps are completed. For each edge \( e \) in the fragment graph \( G \) a counter \( b(e) \) is initialized to be the number of times it occurs in both \( P' \) and \( P'' \). Clearly, \( b(e) \) can only take the values 0, 1 and 2. Instead of using the original edge costs \( c_e \) we now use:
\[
\tilde{c}_e \leftarrow p_{b(e)} c_e
\]
Here, \( p_{b(e)} \) is a perturbation factor. Based on the perturbed costs \( \tilde{c} \) the assembling phase is executed (both the greedy part and the local search). The result is turned into a solution for the fragment graph by restoring the original costs \( c \). In order to make the new solution mimicking its parents \( P' \) and \( P'' \), the perturbation factors should be chosen such that \( p_0 > p_1 > p_2 \), since the lower-weight edges are more likely to end up as a cut-edge. Delling et al. observed that the algorithm is not too sensitive to the choice of parameters. They used \( p_0 = 5, p_1 = 3 \) and \( p_2 = 2 \).

We implemented the PUNCH algorithm and used the parameters suggested by the authors [50]. We did not implement the meta-heuristics, since the resulting solutions after local search are already good enough for our purpose. In Table 6.4 some characteristics are showed for several values of \( n \), the maximum cell size that is allowed in the partition. The table shows the size of the fragment graph, the number of cells in the partition and the size of the core graph that is found. Finally, the number of shortcuts that should dynamically be added to the core graph in real time are mentioned.

<table>
<thead>
<tr>
<th>( n )</th>
<th>Fragment Graph</th>
<th>Partitioning</th>
<th>Final core graph</th>
</tr>
</thead>
<tbody>
<tr>
<td>(</td>
<td>V</td>
<td>)</td>
<td>(</td>
</tr>
<tr>
<td>32,768</td>
<td>106,622</td>
<td>181,032</td>
<td>265</td>
</tr>
<tr>
<td>65,536</td>
<td>69,369</td>
<td>121,459</td>
<td>133</td>
</tr>
<tr>
<td>131,072</td>
<td>42,739</td>
<td>77,228</td>
<td>67</td>
</tr>
<tr>
<td>262,144</td>
<td>25,197</td>
<td>47,197</td>
<td>34</td>
</tr>
<tr>
<td>524,288</td>
<td>13,930</td>
<td>26,894</td>
<td>17</td>
</tr>
</tbody>
</table>

Figure 6.15 visualizes the cut-edges for some different values of \( n \). In part (d) one can clearly see that the border edges between The Netherlands and Germany are found to be natural cuts. The same holds for the rivers in The Netherlands.

As stated before, finding a partition based on natural cuts is the first step of hybrid preprocessing. The next step that should be executed for each new set of arc costs
consists of calculating and storing the shortcuts between the border nodes of each cell in the partition. The memory needed to store a shortcut is only 20 bytes. Hence, the additional data needed for a specific set of arc costs varies from 6Mb ($n = 512\text{K}$) to 17Mb ($n = 32\text{K}$).

![Final partition for several values of $n$ (K = 1,024).](image)

During our experiments we found out that the calculation of shortcuts between the border nodes of each cell can easily be speeded up by (temporarily) removing the so-called quarters from the cell (see Section 6.2.1). Although this reduction step is already used during the filtering-phase, it makes sense to store a flag for each node that is not in the largest biconnected component of the graph. During the filtering phase, every other component is contracted together with is neighbor in the largest biconnected component. Therefore, each cut-edge is definitely part of that largest biconnected component and hence, all other components (quarters) can be safely removed from the cells.
6.5 Hybrid graph preprocessing

Recently, Schulz and Sanders [172, 177] presented a novel graph partitioning algorithm called ‘Karlsruhe Fast Flow Partitioner Evolutionary’ (KaFFPaE). Their algorithm is a so-called evolutionary algorithm (hence, a meta heuristic). When applied to the fragment graph that results after the filtering phase (Figure 6.13), they were able to find comparable or better solutions than the ones found by PUNCH.

6.5.2 Arc flags

Lauther [129] introduced the so-called edge-flags (nowadays known as arc-flags). Given a partitioning \( V = (V^1, V^2, \ldots, V^K) \) there are \( K \) bits stored with each arc. The \( K \) bits together are called the ‘flag vector’. For \((v, w) \in A\), the \( k^{th} \) bit (\( 1 \leq k \leq K \)) in this flag vector is 1 if and only if there exists at least one shortest path from node \( v \) to any node \( u \in V^k \) that contains \((v, w)\). The extra amount of data needed to store the arc flags is bounded by \(|A|K\) bits. Hilger et al. [105] observed that the actual number of different arc flag vectors can be much smaller than \(|A|\). Instead of storing an arc flag vector with each arc, the arc flag vectors are stored in one place and only a pointer is added to each arc. In this way, the memory usage can be reduced.

Before we describe how arc flags can be determined during a preprocessing step, we first show how they can be used in the Generic Shortest Path algorithm. In line 6 of Algorithm 6.18 it is checked whether or not the outgoing arcs of node \( v \) should be scanned at all. If Algorithm 6.18 is implemented as a label-setting method, this check can be omitted and replaced by the stop condition: stop as soon as \( v = t \).

```
Algorithm 6.18: Generic Shortest Path (arc flags) \((G, s, t)\)

Require: A partitioning \( V = (V^1, V^2, \ldots, V^K) \) and arc flag vectors

1: INITIALIZE-SINGLE-SOURCE\((G, s, Q)\)
2: Let \( k \) be the index such that \( t \in V^k \)
3: repeat
4: Select a pivot node \( v \) from \( Q \)
5: \( Q \leftarrow Q - \{v\} \)
6: if \( d_v < d_t \) then
7: foreach \((v, w) \in \delta^+(v)\) do
8: if \( k^{th} \) bit in flag vector of \((v, w)\) is 1 then
9: LABEL-UPDATE\((v, w, Q)\)
8: until \( Q = \emptyset \)
```

The only difference with the Generic Shortest Path algorithm (Algorithm 3.4) is the extra check in line 8. The LABEL-UPDATE\((v, w, Q)\) is not executed if the \( k^{th} \) bit of the arc flag vector of \((v, w)\) is zero. However, this does not influence the correctness of Algorithm 6.18 as one can easily see by the definition of the arc flag vectors. Suppose that a shortest \( s-t \) path contains \((v, w)\) and \( t \in V^k \). By definition, the \( k^{th} \) bit of the arc flag vector of \((v, w)\) is 1. Figure 6.16 [105] shows an example of the use of an arc flag.
In [105] the trade-off between the required space to store the precalculated arc flags and the reduction of the solution space is described as follows. If the partitioning of \( V \) contains only one cell, then all arc flag vectors contain just one bit with value 1 and the solution space is not reduced at all. The solution space is reduced the most if \( V \) is partitioned into \( |V| \) cells each containing one node. Clearly, the required space to store the arc flags is \( O(|V||A|) \) bits. According to [105], the achieved speedups are already enormous for a partitioning \( V = (V_1, V_2, \ldots, V_K) \) with \( K \ll |V| \), while only little extra space is required.

In a bidirectional algorithm, one could use the arc flag vectors that were originally determined for the unidirectional search only. But then the backward search that origins from \( t \) is not ‘guided’ towards the cell that contains \( s \). Hence, if a bidirectional approach is used, one should use two arc flag vectors for each arc. The \( k^{th} \) (\( 1 \leq k \leq K \)) bit in the extra flag vector of \((v,w)\) is 1 if and only if there exists a shortest path that starts at a node \( u \in V_k \) that contains \((v,w)\). Theoretically, one can use two different partitions, one for the forward search and one for the backward search.

### 6.5.2.1 Determining the arc flags

In order to determine the arc flag vectors one could initialize all arc flag vectors as a \( K \) bit zero vector. By running the Generic Shortest Path algorithm \(|V|\) times on the reverse graph, the shortest path tree to node \( t \) is determined once for each node \( t \in V \). Let \( V^k \) be the cell such that \( t \in V^k \). For each arc \((v,w)\) in the shortest path tree rooted at \( t \) the flag vector is updated such that the \( k^{th} \) bit is set to 1. Lauther [129] observed that instead of \(|V|\) shortest path trees one only has to focus on boundary nodes. Recall that a boundary node is a node that is adjacent to a boundary edge (see Definition 6.31). For each shortest \( s-t \) path with \( s \in V^i \), \( t \in V^j \) and \( i \neq j \) the path contains at least one boundary node of cell \( V^i \). Arc flag vectors can be calculated for each cell \( V^k \) as follows. For any arc \((v,w)\) with \( v \in V^k \) and \( w \in V^k \) the \( k^{th} \) bit of the arc flag vector is set to 1. For each boundary node \( t \) of cell...
$V^k$, a shortest path tree rooted at $t$ is determined in the reverse graph. For each arc in such a shortest path tree, the $k^{th}$ bit of the arc flag vector is set to 1.

Lauther already observed that the idea to contract all nodes of a cell and only grow a shortest path tree of the contracted node is equal to initializing the Generic Shortest Path algorithm with all border nodes of that cell and distance zero. However, the resulting arc flags vectors are now based on only the nearest node inside the cell instead of any node of the cell. In that case, an $s$-$t$ path will be found that goes from $s$ to its nearest border node $b$ of the cell that contains $t$. The $s$-$t$ path will continue via a shortest $b$-$t$ path. However, this is not necessarily a shortest $s$-$t$ path.

Hilger et al. [105] observed that one only has to determine the arc flags for the largest biconnected part (the ‘core’) of the graph. Since we assume that the graph is strongly connected, every non-core node is part of a quarter (see Section 6.2.1). Each quarter has exactly one adjacent core node $v$. All nodes in a quarter are assigned to the same cell $V^k$ as node $v$. The forward arc flag vector of any arc in that quarter has only the $k^{th}$ bit set to 1, since only nodes in the quarter can be reached via that arc. All bits in the backward arc flag vector are set to 1, since these arcs form, most likely, the first part of a shortest path from a node in the quarter to any other node in the graph. If the graph partitioning is used that is described in Section 6.5.1, all nodes in one quarter already belong to a same cell. However, by considering only the non-core nodes the number of nodes in the shortest path trees that are built can be reduced by about 29% (see Table 6.2). If we bypass the degree 2 nodes, by running Algorithm 6.1, and ensuring that for each newly created shortcut its adjacent nodes are in the same cell of the partition that is considered, a similar reasoning can be done for the arc flag vectors of the original arcs of that shortcut. From Table 6.2 it can readily be seen that the shortest path trees in that case only contains 31% of the original amount of nodes.

There are possibilities to reduce the time needed for calculating these shortest path trees. Lauther [129] observed that for any two border nodes of a cell the shortest path trees rooted at these border nodes in the reverse graph will usually show much similarity. He suggests to exploit these similarities, without providing any detail on how this can be done. Hilger et al. [105] mention two ways to exploit these similarities: either by using ‘bit-recycling’ or by using ‘centralized shortest paths’. We will briefly describe both methods.

In the ‘bit-recycling’ approach, the border nodes $b_1, b_2, \ldots, b_n$ of cell $V^k$ are ordered by the sequence in which they occur when one walks along the boundary of that cell. During the calculation of the shortest path tree rooted at $b_i$ $(2 \leq i \leq n)$ in the reverse graph the shortest path tree rooted at $b_{i-1}$ is used. Each time a pivot node $d_v$ is selected during the calculation of the shortest path tree rooted at $b_i$, good upper bounds are obtained by traversing the shortest path tree rooted at $b_{i-1}$ starting at node $v$ and propagating the label $d_v$. In [105] it is observed that bit-recycling does not lead to a considerable speedup if applied on a roadmap graph, but it might be efficient on more dense graphs.
The ‘centralized-shortest path’ approach uses a multiple label $D_v = (d_1^v, d_2^v, \ldots, d_n^v)$ for each node $v \in V$. Here, $d_i^v$ equals the length of a path from border node $b_i$ of a certain cell (with border nodes $b_1, b_2, \ldots, b_n$) to node $v$ in the reverse graph. Each time a pivot node $v$ is selected, its outgoing arcs $(v, w) \in \delta^+(v)$ in the reverse graph, are scanned and each entry of label $D_w$ is updated: $d_i^w \leftarrow \min\{d_i^w, d_i^v + c_{vw}\}$ for all $1 \leq i \leq n$. The label $D_v$ can be initialized by setting all entries to infinity except for $d_i^{b_i}$ ($1 \leq i \leq n$) that are initialized to zero, since they represent the distance from a border node $b_i$ to itself. Beside a multiple label $D_v$ a key value $k_v$ is assigned to each node $v \in V$. In each iteration a node with minimal $k_v$ is chosen from the set of candidate nodes $Q$. In [105] two possible keys are proposed: the minimum tentative key and the minimal total key. The minimum tentative key is maintained as follows.

Each time an arc $(v, w)$ is scanned, the set $K$ is determined such that $K$ contains all entries of $D_w$ that are altered during the scan. If $w \in Q$, then $k_w \leftarrow \min(K \cup \{k_w\})$. In case $w \notin Q$, $k_w \leftarrow \min(K)$. In [105] it is proved that any node will enter $Q$ at most $n$ times if a minimum tentative key is used. The minimum total key is defined as the minimum entry in a label $D_w$, regardless whether or not this entry is updated. Although this might lead to more iterations, the minimum total key usually outperforms the minimum tentative key, as observed by Hilger et al.

6.5.2.2 Choosing a partition

The speedup that is achieved with arc flags is highly dependent on the graph partition that is used. Möhring et al. [136] provide an experimental evaluation on several graph partitions. Originally, Lauther [129] used a rectangular (grid) partition of the graph. He also mentioned a somewhat more sophisticated partition based on so-called square covering. In [136] experiments are done with a grid, a so-called $kd$-tree and METIS. The graph partitioning algorithm METIS is presented by Karypis and Kumar [118]. It is a well-known algorithm that creates a partition with almost equal sized cells and a small set of boundary nodes. However, it is outperformed by far by the PUNCH method of Delling et al. [50] described above. In [105] the following objectives that should be fulfilled by a graph partition are mentioned:

- The number of boundary arcs (and hence boundary nodes) should be small, since the preprocessing time depends directly on this number.
- All cells in the partition should be more or less equally sized. In that case, each bit in the arc flag vector is ‘responsible’ for a similar amount of possible target nodes.
- The number of almost full flag vectors should be small. Clearly, a full arc flag vector means that the arc can never be excluded.

At least the first two conditions are met with PUNCH. The last one can only be verified after determining the actual arc flags. Clearly, determining the graph partition is done once, independently of the arc costs. In this way, the use of arc flags can be seen as a hybrid method. However, in a hybrid method the second step that is based on actual arc costs must be executed in a few seconds and add only a small amount
of data to the graph. Determining the arc flags is rather time consuming: a shortest path tree has to be determined for each border node in the partition. Therefore, research is done into how arc flags can be updated if (some) arc costs are changed.

6.5.2.3 Recomputing the arc flags

Berretini et al. [16] made a first attempt to use arc flags in a dynamic environment. They only focused on situations where the arc costs increase. In practical situations this is exactly what is happening. In case of a traffic jam, or during rush hours the travel time on several arcs is increased. The question is how the initial arc flags (based on the ‘regular’ arc costs) can be updated without computing them from scratch. A threshold for each arc $(v, w)$ is determined and used in the following way. If an arc cost is increased, the increase is compared to the threshold. In this way, it is determined whether or not $(v, w)$ becomes a starting arc of a shortest path to a boundary node. In the latter case, the bit of the arc flag vector of $(v, w)$ that corresponds to the cell of that border node, is set to 1. In this way, the correctness of Algorithm 6.18 is ensured. The drawback of this update procedure is that a bit that is set to 1 will never be reset to 0. This might lead to an efficiency loss of the algorithm. Especially when the arc costs are changed (increased) several times, this defect will grow. Details in how to determine and use these thresholds can be found in [16].

D’Angelo et al. [38] introduced a data structure called ‘Road-Signs’. Road-Signs can be used to compute arc flags. They can be used for efficiently recalculating the arc flags in case there are only increased arc costs. Compared to Berretini et al. no bit remains unnecessarily at value 1. Hence, using Road-Signs will not lead to an efficiency loss at all. Road-Signs can be used in a dynamic environment where the arc costs are changed (increased) more than once. In [37] D’Angelo et al. were able to improve the Road-Signs data structure such that it can be used for decreasing arc costs as well. Hence, it is possible to update arc flags in a reasonable amount of time.
Computational results

In this chapter we present the results of several experiments, where we compare optimization algorithms, heuristic estimators, preprocessing techniques and different alternation rules for bidirectional search. In Section 7.1 we describe how the experiments are designed. Sections 7.3–7.6 present results of experiments on graphs that are preprocessed. We draw some conclusions in Section 7.7.

7.1 Experimental setup

We use the combined road map of Germany, the Netherlands, Belgium and Luxembourg, provided by TomTom (Multinet, July 2010) as a basis for input. See Section 6.1.1 for more details about this road map and how a directed graph $G_{car} = (V, A)$ can be constructed from it. From that graph, we extract the strongly connected subgraph $G_{car} = (V, A)$ with $|V| = 8,678,011$ and $|A| = 19,572,341$ by running the StrongConnect procedure of Tarjan [184]. In this chapter, we call $G_{car}$ the ‘full graph’. We run several experiments on this full graph and we apply different preprocessing techniques to it.

Since in practical situations we are especially interested in fastest paths, the arc cost $c_{vw}$ represents the travel time to travel from node $v$ to node $w$ along the arc $(v, w)$. The arc costs are integer. The reason that we use integer arc costs is twofold. Integer arc costs can be stored in half the amount of memory needed to store them as real numbers. Secondly, some data structures that implement the priority queue assume that the priority keys are integer. As already mentioned in Section 2.2.7, considering the arc costs to be integer is not restrictive for road map graphs, since they can be multiplied by a suitably large number, before rounding. We experienced some problems when the arc costs were rounded to seconds. It happened that the shortest path went via the parking place along a highway instead of the highway itself, due to the fact that the driving time via the parking place was equal to the driving time via the highway. By expressing the arc costs in tenth of seconds, these
undesired results do not occur in our experiments.

Table 7.1 shows some characteristics of the arc cost distribution. The maximum arc cost amounts to two days. This very high maximum is due to some long ferry lines in the graph, that have an assigned average speed of 3km/h. If we drop these ferry lines, the remaining arc costs are all less than 18,000 (half an hour).

<table>
<thead>
<tr>
<th>Minimum</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>25%-percentile</td>
<td>57</td>
</tr>
<tr>
<td>Median</td>
<td>120</td>
</tr>
<tr>
<td>75%-percentile</td>
<td>228</td>
</tr>
<tr>
<td>Maximum</td>
<td>1,763,685</td>
</tr>
</tbody>
</table>

All algorithms in this chapter are implemented in Delphi XE3. Running times that are reported are based on a Dell Precision T3600 with a 3.6 Ghz processor and 16 Gb memory.

7.1.1 Comparison of optimization algorithms

We compare the following optimization algorithms:

- Dijkstra implemented with a heap (7 variants)
- Dijkstra implemented with a bucket data structure (7 variants)
- Bellman-Ford-Moore (4 variants)
- d’Esopo-Pape (2 variants)
- Pallottino (2 variants)
- Small Label First (4 variants)
- Threshold algorithm
- Approximate bucket method

Hence, a total of 28 optimization algorithms are implemented.

We implemented Dijkstra in 14 ways. Each implementation uses a different implementation of the priority queue $Q$, seven of them are heap-based and the others are bucket-based. In the tables of this chapter, we denote Dijkstra shortly with the acronym LS for label-setting. We implemented the following heaps: a $d$-heap, for $d = 2, 3, 4, 5$. Furthermore, the Binomial heap, the Fibonacci heap and the Relaxed heap are implemented. Since we use integral arc costs, we also implemented several bucket data structures: Dial (with an overflow bag), the so-called $k$-level buckets ($k = 2, 3, 4, 5, 6$) and the radix heap. In Appendix B all these data structures are discussed in detail.

Based on the arc cost distribution (see Table 7.1) we used the following parameters in our bucket implementations: Dial’s method is implemented with 2,048 buckets.
and an overflow bag. The $k$-level buckets are implemented for $k = 2, 3, 4, 5, 6$ with $b$ buckets at each level, where $b$ is chosen such that the total range of the bucket data structure is at least 252,000 (7 hours). An overflow bag is added for possible larger label values. As soon as the buckets are all emptied, the overflow bag is redistributed over the buckets in a similar way as the overflow bag of Dial’s method. See Table 7.2 for more details about the $k$-level bucket implementations we used. Note that the maximum label that can be stored in a $k$-level bucket data structure with $b$ buckets at each level is $b + b^2 + \ldots + b^k - 1$.

**Table 7.2: Range of $k$-level buckets with $b$ buckets at each level**

<table>
<thead>
<tr>
<th>$k$</th>
<th>$b$</th>
<th>range</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>512</td>
<td>0 – 262,655</td>
</tr>
<tr>
<td>3</td>
<td>64</td>
<td>0 – 266,303</td>
</tr>
<tr>
<td>4</td>
<td>24</td>
<td>0 – 346,199</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
<td>0 – 271,451</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>0 – 299,591</td>
</tr>
</tbody>
</table>

As a comparison between label-setting and label-correcting algorithms we implemented the following label-correcting algorithms. **Bellman-Ford-Moore** is implemented in its basic form, but also enriched with parent-checking (Section 3.3.3.2) and Large Label Last (Section 3.3.3.3) or both. In the tables below we abbreviated **Bellman-Ford-Moore** to BFM, Large Label Last to LLL and parent-checking to PC. We implemented d’Esopo-Pape and Pallottino in their basic forms and also in a version that uses the Large Label Last strategy. We did not apply parent-checking to these algorithms, since they are not expected to gain from parent-checking, which we experimentally verified already in Klunder and Post [121]. We implemented **Small Label First** in the same four variants as **Bellman-Ford-Moore**. We implemented the Threshold algorithm (Section 3.3.2.1) in a version that divides the priority queue $Q$ in two distinct queues $Q'$ and $Q''$. Nodes are added to $Q'$ if their node label exceeds a certain threshold $\tau$ and to $Q''$ otherwise. A node that is deleted from $Q$ is always selected from $Q'$. If $Q' = \emptyset$ and $Q'' \neq \emptyset$, then the threshold $\tau$ is updated as follows: $\tau \leftarrow \min\{|d_v| v \in Q''\} + 512$ and all nodes of $Q''$ with $d_v \leq \tau$ are transferred to $Q'$. The last label-correcting algorithm we implemented is called **Approximate buckets** (see Section B.2.3) and is motivated by the success of Dial’s bucket implementation. In our implementation this method uses 800 buckets with a bucket width of 512. If a node $d_v$ has to be stored with a label that exceeds $800 \times 512$, the node is placed in the highest bucket. If landmarks are used, the approximate bucket method is initialized with 12,800 buckets, each with a width of 32.

### 7.1.2 Estimators that are compared

The effect of the use of estimators is tested, by running all 28 optimization algorithms mentioned above in three ways:

- without an estimator,
with a Euclidean estimator,
- with a landmark estimator.

For the Euclidean estimator (5.7) we make use of the geographical coordinates of each node. Although, formally $A^*$ is a family of algorithms rather than an estimator, commonly a member of the $A^*$ family is instantiated by the choice of a Euclidean estimator (see Section 5.1.3). We therefore use $A^*$ in the tables below, to actually denote this particular member of the $A^*$ family and to distinguish it from another member of the family that uses the landmark estimator. The latter one is denoted by ‘Landmark’ in this section. For the landmark estimator we use a set of 64 landmarks determined by the so-called ‘maxcover’ heuristic (see Figure 6.9). At first, for each specific $s$-$t$ shortest path calculation, the four landmarks that provide the highest lower bounds on $d(s,t)$ are chosen as active landmarks. However, as we will see below, our active landmark selection criterium as stated in Section 6.3.3.2 provides better results. Hence, we changed to that criterium after showing why it leads to better results.

### 7.1.3 Bidirectional methods

Bidirectional methods are tested with three alternation rules:

- **Dantzig**: Alternate between forward search and backward search.
- **Pohl**: Choose the direction for which the cardinality of the candidate set $Q$ is minimal.
- **Nicholson**: Choose the direction for which the top node of the candidate set $Q$ has the smallest label.

The original alternation rule of Nicholson chooses the direction for which the node with minimum label in the candidate set $Q$ has the smallest label. Here, we modified the alternation rule, such that it can also be applied to label-correcting algorithms.

We implemented bidirectional search in the following ways:

- without an estimator,
- with a balanced Euclidean estimator,
- with a symmetric Euclidean estimator, by implementing our new bound, see Proposition 5.20,
- with a symmetric Euclidean estimator, using scalar projections,
- with a balanced landmark estimator.

Recall that our new bound can only be applied to label-setting algorithms. As we will see below, the landmark estimator performs badly when implemented in a symmetric way. Therefore, we did not implement the landmark estimator in a symmetric way, neither in combination with our new bound, nor combined with scalar projection. But we will provide some insight into the reason why a landmark estimator can better be implemented in a balanced setting.
7.1 Experimental setup

7.1.4 Preprocessing techniques

We ran experiments on the full graph and on preprocessed versions of that graph. We implemented the following preprocessing techniques:

- static preprocessing,
- dynamic preprocessing based on node and arc reduction,
- hybrid preprocessing,
- contraction hierarchies.

7.1.5 Test set

Our test set consists of 100 randomly chosen pairs of origins and destinations. They are visualized in Figure 7.1. Characteristics of the distribution of the shortest path lengths are provided in Table 7.3.

![Figure 7.1: Test set.](image)

<table>
<thead>
<tr>
<th>Percentile</th>
<th>Path-cost</th>
<th>Travel time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Minimum</td>
<td>13,539</td>
<td>0:23</td>
</tr>
<tr>
<td>25%-percentile</td>
<td>122,914</td>
<td>3:25</td>
</tr>
<tr>
<td>Median</td>
<td>168,462</td>
<td>4:41</td>
</tr>
<tr>
<td>75%-percentile</td>
<td>239,833</td>
<td>6:40</td>
</tr>
<tr>
<td>Maximum</td>
<td>350,565</td>
<td>9:44</td>
</tr>
</tbody>
</table>

Table 7.3: Path length distribution of our test set
7.2 Experiments on the full graph

In this section we present the results of several experiments on the full graph. Although we will see in later sections that the use of any preprocessing technique will provide far better results, we use these experiments as a yardstick to compare the effectiveness of the chosen preprocessing techniques. Beside that, we provide some general observations that seem to occur in all experiments.

Furthermore, we encountered issues that will not take any advantage of the preprocessing techniques described here. For example, in Emergency Vehicle Routing problems the area that can be reached within a certain amount of time should be determined. This means the partial growing of a shortest path tree, until the time bound is reached. At the end of this section we determine the area that can be reached from any origin in our test set within 10, 20, 30 and 40 minutes to provide some insight in the effectiveness of the algorithms that are discussed.

It turns out that, based on our results, we were able to provide an improvement when using the landmark estimator in a unidirectional way. In Section 7.2.1.1 we describe this improvement in more detail and we will incorporate this improvement in our tests on the preprocessed graphs as well, to verify if the improvement is effective on that graphs as well.

7.2.1 Unidirectional search on the full graph

Table 7.4 shows the cumulated running times and iterations for each algorithm, when unidirectional search is applied.

Before we highlight some general observations, we will first discuss the somewhat strange results in the landmark column. One might expect that the number of iterations is about the same for each specific implementation of the priority queue $Q$ in a label setting environment. They might differ slightly, since we did not implement a tie-breaking rule as suggested by Nilsson [141]: if there are multiple nodes with minimum label in $Q$ and the destination node $t$ is one of them, $t$ should be chosen.

There are two reasons for not implementing this simple rule:

- We implemented $Q$ as an abstract data type (ADT), that has no knowledge about the destination node.
- The overhead of checking whether or not $t$ is one of the nodes with minimum label in each iteration turns out to take more time than just executing some extra iterations.

From Table 7.4 it can readily be seen that for label-setting methods that use no estimates and for methods that use Euclidean ($A^*$) estimates, the number of iterations needed is indeed about the same for each specific implementation of the priority queue. However, when a landmark estimate is used, there is a significant difference in the number of iterations between for example the 4-heap implementation and the
### Table 7.4: Unidirectional methods on the full graph. Cumulated running times (seconds) and iterations (in millions). ‘Landm. dfs’ stands for the landmark estimator where an initial Depth First Search is executed. This method will be explained later.

<table>
<thead>
<tr>
<th>Method</th>
<th>No estimator</th>
<th>A*</th>
<th>Landmarks</th>
<th>Landm. dfs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iter.</td>
<td>CPU</td>
<td>Iter.</td>
<td>CPU</td>
</tr>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>440.1</td>
<td>336.8</td>
<td>137.9</td>
<td>125.9</td>
</tr>
<tr>
<td>3-heap</td>
<td>440.1</td>
<td>327.8</td>
<td>137.9</td>
<td>123.3</td>
</tr>
<tr>
<td>4-heap</td>
<td>440.1</td>
<td>326.9</td>
<td>137.9</td>
<td>122.4</td>
</tr>
<tr>
<td>5-heap</td>
<td>440.1</td>
<td>331.2</td>
<td>137.9</td>
<td>123.6</td>
</tr>
<tr>
<td>Binomial</td>
<td>440.1</td>
<td>409.4</td>
<td>137.9</td>
<td>151.0</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>440.1</td>
<td>464.0</td>
<td>137.9</td>
<td>167.1</td>
</tr>
<tr>
<td>Relaxed</td>
<td>440.1</td>
<td>517.5</td>
<td>137.9</td>
<td>188.7</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>440.1</td>
<td>202.2</td>
<td>137.9</td>
<td>81.3</td>
</tr>
<tr>
<td>2-level 512</td>
<td>440.1</td>
<td>216.4</td>
<td>137.9</td>
<td>85.1</td>
</tr>
<tr>
<td>3-level 64</td>
<td>440.1</td>
<td>228.7</td>
<td>137.9</td>
<td>90.3</td>
</tr>
<tr>
<td>4-level 24</td>
<td>440.1</td>
<td>235.3</td>
<td>137.9</td>
<td>93.5</td>
</tr>
<tr>
<td>5-level 12</td>
<td>440.1</td>
<td>247.7</td>
<td>137.9</td>
<td>95.8</td>
</tr>
<tr>
<td>6-level 8</td>
<td>440.1</td>
<td>251.1</td>
<td>137.9</td>
<td>98.4</td>
</tr>
<tr>
<td>radix</td>
<td>440.1</td>
<td>280.0</td>
<td>137.9</td>
<td>110.6</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>22183.0</td>
<td>9407.2</td>
<td>18722.8</td>
<td>9173.8</td>
</tr>
<tr>
<td>Parent check</td>
<td>10674.1</td>
<td>5037.0</td>
<td>8986.9</td>
<td>4879.6</td>
</tr>
<tr>
<td>LLL</td>
<td>578.4</td>
<td>252.1</td>
<td>232.0</td>
<td>127.9</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>541.7</td>
<td>261.1</td>
<td>210.5</td>
<td>131.1</td>
</tr>
<tr>
<td>d'Esopo-P.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>3038.3</td>
<td>556.6</td>
<td>2944.7</td>
<td>657.8</td>
</tr>
<tr>
<td>LLL</td>
<td>507.5</td>
<td>221.2</td>
<td>211.0</td>
<td>110.9</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>1181.6</td>
<td>360.0</td>
<td>1134.8</td>
<td>430.2</td>
</tr>
<tr>
<td>LLL</td>
<td>493.0</td>
<td>220.7</td>
<td>186.0</td>
<td>108.1</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>2344.9</td>
<td>1029.1</td>
<td>742.5</td>
<td>377.1</td>
</tr>
<tr>
<td>Parent check</td>
<td>2026.9</td>
<td>939.5</td>
<td>628.8</td>
<td>338.9</td>
</tr>
<tr>
<td>LLL</td>
<td>478.2</td>
<td>209.8</td>
<td>156.0</td>
<td>91.2</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>470.7</td>
<td>218.8</td>
<td>151.7</td>
<td>93.4</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>444.4</td>
<td>219.6</td>
<td>139.4</td>
<td>88.5</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>479.6</td>
<td>186.9</td>
<td>151.6</td>
<td>72.3</td>
</tr>
</tbody>
</table>
radix heap implementation. Therefore, we tested whether or not the addition of a tie-breaking rule might reduce this difference. It turns out that even if such a tie-breaking rule is used, different implementations of the priority queue might need a different number of iterations. The number of iterations needed to label all nodes \( v \in \mathcal{V} \) with \( d(s,v) < d(s,t) \) clearly is the same for each specific implementation of the priority queue. However among all nodes \( v \in \mathcal{V} \) with \( d(s,v) = d(s,t) \) there is no predetermined order in which the label \( d_v \) gets its final value \( d(v,t) \).

### 7.2.1.1 Improvement on unidirectional landmark-based methods

We have the following two observations:

1. The number of iterations needed by label-setting algorithms differs for each specific implementation of the priority queue \( Q \).
2. The number of iterations needed to determine the final node labels \( d_v = d(s,v) \) for all nodes \( v \) with \( d(s,v) < d(s,t) \) does not depend on the specific implementation of the priority queue provided that a label-setting algorithm is used.

As soon as all nodes \( v \) with \( d(s,v) < d(s,t) \) have reached their final label value, a tie-breaking rule as mentioned above should try to pick node \( t \) as the next candidate node. However, it might happen that node \( t \) is not reachable yet, since multiple arcs might have arc cost zero. From Theorem 5.13, we know that using a consistent heuristic estimator \( h \) is equivalent to running Dijkstra with the modified arc costs \( c'_{vw} = c_{vw} + h_v - h_w \). Especially when a landmark estimator \( h \) is used, several arcs get a modified arc cost \( c'_{vw} = 0 \). The number of arcs that has a zero modified arc cost is in fact a measurement of the quality of the landmark set (Section 6.3.3.1). Hence, it might take a while before node \( t \) is actually reached, no matter if a tie-breaking rule is used or not.

We implemented a way to overcome this drawback as follows. After selecting a fixed set of active landmarks, a graph traversal is started from \( t \) in the reverse graph. Only arcs with zero modified arc cost are taken into account. Each node \( v \) that can be reached from node \( t \) is added to a set of destination nodes \( \mathcal{T} \). We add a backward parent \( \pi'_v \), to easily restore the path from any node \( v \in \mathcal{T} \) to the original destination node \( t \). Algorithm 7.1 states this procedure in a more formal way. This procedure takes the original graph as input and \( h \) is the landmark estimator that is based on the set of active landmarks. The candidate set \( Q \) in Algorithm 7.1 can be implemented as an ordinary stack, resulting in a Depth First Search traversal.

The cumulated cardinalities of the \( \mathcal{T} \)-sets adds up to 3,476,621. Hence, we might expect that the running times will be much better if a set of destinations \( \mathcal{T} \) is used instead of only one node \( t \). Although one might argue that, strictly spoken, the determination of \( \mathcal{T} \) needs a reverse graph and therefore operates in a bidirectional way, there is actually no shortest path search in the backward graph, only a straightforward graph traversal. The last two columns of Table 7.4 shows the results if a Depth First Search from \( t \) is started first. Note that the running times include the
7.2 Experiments on the full graph

Algorithm 7.1: Landmark-graph traversal \( (G = (V, A), h, t) \)

1. \( T \leftarrow \{ t \} \)
2. \( \pi_T \leftarrow \text{NIL} \)
3. \( Q \leftarrow \{ t \} \)
4. While \( Q \neq \emptyset \) do
5. Select a pivot node \( w \) from \( Q \)
6. \( Q \leftarrow Q - \{ w \} \)
7. For each \( (v, w) \in \delta^{-}(w) \) do
8. If \( v \notin T \) and \( c_{vw} + h_w - h_v = 0 \) then
9. \( T \leftarrow T \cup \{ v \} \)
10. \( \pi_T \leftarrow w \)
11. Return \( T, \pi_t \)

Time needed for this Depth First Search. Hence, they are comparable with the other running times of Table 7.4. It is immediately clear that the number of iterations of label setting algorithms now are about equal. Furthermore, a speedup of 35%–80% is realised. The smallest gain was found in the bucket-based Dijkstra variants, and the highest in the label correcting methods.

We analysed the effect of adding a Depth First Search in more detail. Typically, one of the nodes in \( T \) will be very close to the origin node \( s \). Figure 7.2 illustrates such an example. One could imagine that growing a search tree from the origin node (the red square left-below) will hit a destination node (a blue dot) very soon.

Figure 7.2: Typical behaviour. The red squares are the origin and destination. The blue dots are the nodes in \( T \) and the black diamonds are the four active landmarks (one of them is partially overlapped by the upper red square).
We encountered two types of issues when running an initial Depth First Search. First, it might happen that $|T|$ becomes very big. An example is illustrated in Figure 7.3. In this example, $|T| = 3,118,095$, thus taking about 90% of the cumulated cardinalities mentioned before. Clearly, we can overcome this behaviour by terminating Algorithm 7.1 as soon as $|T|$ reaches a certain bound, for example 5,000.

![Figure 7.3](image)

Figure 7.3: Two bad $T$-sets. Too big (left) and wrong direction (right).

The other problem arises when the active landmarks are lying close together and not in the direction of the source node at all. Instead of sorting the landmarks on $h_L(s)$, the initial lower bound for $d(s,t)$ based on landmark $L$, and declaring the four landmarks that provide the highest bounds to be active, we can use other selection criteria. In Section 6.3.3.2 we mentioned our selection criterium [121]: Four landmarks $L_1, L_2, L_3$ and $L_4$ are chosen such that $L_1$ maximizes the lower bound for $d(s,t)$, $L_2$ lies nearby $s$, $L_3$ lies nearby $t$ and $L_4$ is somewhere in the middle of $s$ and $t$. Figure 7.4 shows the effect of this strategy for the bad $T$ set encountered in the right part of Figure 7.3.

In Table 7.5 we show the results for the unidirectional label-setting algorithms if our active landmark selection criterium is used. We did not include the results for the label-correcting methods here, since they could not compete with the label-setting variants as can readily be seen from Table 7.4. From here on, we switch to our active landmark selection criterium, instead of only taking the four landmarks that provide the highest lower bounds. Even without applying an initial Depth First Search to determine a set of destination nodes, the running times already decrease with 6%–28%. If the initial Depth First Search is applied, our selection criterium reduces the running times with 66%–77%.
Figure 7.4: *Our active landmark selection criterium provides a good $T$ set.*

Table 7.5: *Unidirectional landmark estimator on the full graph, using our active landmark selection criterium. Cumulated running times (seconds) and iterations (in millions).*

<table>
<thead>
<tr>
<th>Method</th>
<th>Landmarks</th>
<th>Landm. dfs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iter.</td>
<td>CPU</td>
</tr>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>42.1</td>
<td>28.4</td>
</tr>
<tr>
<td>3-heap</td>
<td>43.7</td>
<td>29.3</td>
</tr>
<tr>
<td>4-heap</td>
<td>41.1</td>
<td>27.0</td>
</tr>
<tr>
<td>5-heap</td>
<td>42.7</td>
<td>28.4</td>
</tr>
<tr>
<td>Binomial</td>
<td>37.2</td>
<td>27.5</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>39.7</td>
<td>31.5</td>
</tr>
<tr>
<td>Relaxed</td>
<td>34.8</td>
<td>34.3</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>32.1</td>
<td>18.1</td>
</tr>
<tr>
<td>2-level 512</td>
<td>32.1</td>
<td>18.6</td>
</tr>
<tr>
<td>3-level 64</td>
<td>32.1</td>
<td>18.6</td>
</tr>
<tr>
<td>4-level 24</td>
<td>32.1</td>
<td>18.8</td>
</tr>
<tr>
<td>5-level 12</td>
<td>32.1</td>
<td>19.0</td>
</tr>
<tr>
<td>6-level 8</td>
<td>32.1</td>
<td>18.9</td>
</tr>
<tr>
<td>radix</td>
<td>32.1</td>
<td>19.0</td>
</tr>
</tbody>
</table>
7.2.1.2 General observations

Based on Table 7.4 we mention some observations that seem to hold for all results presented in this chapter. The heap-based implementations of Dijkstra perform quite similar for all $d$-heap implementations. Although the Fibonacci heap and the Relaxed heap have the best worst-case bounds, they are outperformed by all other heap-based implementations, including the Binomial heap. The Fibonacci heap is often outperformed by other, theoretically less efficient, data structures [23, 35]. The same seems to hold for the Relaxed heap. Any bucket-based implementation of Dijkstra outperforms each heap-based implementation. Especially Dial’s data structure turns out to be very effective. For the bucket-based implementations the more sophisticated $k$-level buckets and the radix heap performs worse than Dial’s method in our experiments.

The potential-invariant algorithms Bellman-Ford-Moore, d’Esopo-Pape and Pallottino are outperformed by Dijkstra with an ordinary binary heap implementation. Golden [97] stated that Bellman-Ford-Moore can be superior to Dijkstra. For graphs of the size of our input graph, this is clearly not the case, however Bellman-Ford-Moore possibly outperforms Dijkstra if $Q$ is implemented as an ordinary linked-list. Adding the parent-check improvement to Bellman-Ford reduces the running time by about 50%. Applying the Large Label Last strategy to Bellman-Ford-Moore makes it superior to the heap-based implementations of Dijkstra. Applying both Large Label Last and parent checking reduces the number of iterations slightly, but this does not outweigh the extra time needed to evaluate the parent check: the running time slightly increases. Although d’Esopo-Pape and Pallottino are quite similar the latter one outperforms d’Esopo-Pape. Hence, Pallottino should be preferred over d’Esopo-Pape not only for its better worst-case bound, but also from empirical experimentations. Both algorithms significantly reduce their running times if the Large Label Last strategy is added. In that case the algorithms perform quite similar. Both, d’Esopo-Pape and Pallottino perform better than Bellman-Ford-Moore in either their basic implementations or combined with Large Label Last.

Small Label First performs somewhere in between of Pallottino and d’Esopo-Pape if no estimator is applied. Bertsekas [18] observed that Small Label First outperformed d’Esopo-Pape and referred to [76] to state that Pallottino and d’Esopo-Pape perform roughly similar in practise. The latter is clearly not the case in our experiments. Applying the parent check to Small Label First improves the effectiveness of the algorithm. When the Large Label Last strategy is applied to Small Label First its running time is less than all label setting algorithms except for Dial’s method. As with Bellman-Ford-Moore, applying both the parent check and the Large Label Last strategy to Small Label First only slightly decreases the number of iterations, but increases the running time.

The Threshold algorithm performs quite well and can compete with most label-setting implementations. The Approximate Buckets algorithm seems to be superior to all other algorithms. But this only holds for situations where no landmarks
are used. Hence, we advise to implement Dijkstra with Dial’s heap, since it is either the fastest algorithm itself in our experiments or close to it. However, Dial’s method needs integer arc costs, while approximate buckets can be applied to real valued arc costs as well. Hence, one should consider the approximate buckets to be a good alternative as well.

Applying a Euclidean estimator to the unidirectional methods roughly reduces the number of iterations of label-setting algorithms with 70% and the running time with 60%. Obviously, the evaluation of an estimate takes some time. As one might expect, the potential-invariant methods only slightly reduce the running times when an estimator (either Euclidean or landmark based) is applied. They only profit from these estimates after an s-t path is found. From that moment on, a pivot node v can be omitted if \( d_v + h_v \geq d_t \).

Adding a landmark estimator to the unidirectional methods strongly improves the running times. For example, the running time of Dijkstra implemented with Dial’s data structure decreases from 202.2 to 8.2 seconds, a reduction of 96%. The landmark estimator is already discussed in detail in Section 7.2.1.1.

### 7.2.2 Bidirectional search on the full graph

Table 7.6 shows the results for bidirectional implementations that use no estimator, for three different alternation rules. Both, the number of iterations and the running times reduce by about 40%. Since the nodes are not uniformly divided over the graph, one might expect that a reduction of 50% cannot be reached. Pohl’s alternation rule turns out to be the most effective in our experiments.

#### 7.2.2.1 Bidirectional search using A*

Table 7.7 shows the results if the optimization algorithms are implemented in a bidirectional way, using a balanced Euclidean estimator.

Compared with the A* columns of Table 7.4 the number of iterations for label-setting algorithms is reduced by about 60% and the running time by about 40%. Note that in a balanced approach the evaluation of a heuristic estimate takes twice as many operations, since both an estimate towards t and an estimate towards s must be determined. Therefore, the reduction of the number of iterations does not imply an equivalent reduction in the running times. Even potential-invariant label-correcting algorithms show a quite similar reduction in both the number of iterations and running times. Again, Pohl’s alternation rule turns out to be the most efficient. Furthermore, the approximate buckets algorithm outperforms all other algorithms in each column.

In Section 5.3.3 we proposed a new symmetric approach, based on a new bound. Table 7.8 shows the results for the bidirectional label-setting algorithms that use this bound. Compared to Table 7.7 the number of iterations and the running times are quite similar. We observed this already in [154]. Let \( \delta(v, w) \) denote the Euclidean
Table 7.6: Bidirectional methods (no estimator) on the full graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>CPU</td>
<td>CPU</td>
<td>CPU</td>
</tr>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>274.3</td>
<td>233.8</td>
<td>263.5</td>
</tr>
<tr>
<td>3-heap</td>
<td>274.3</td>
<td>227.5</td>
<td>263.5</td>
</tr>
<tr>
<td>4-heap</td>
<td>274.3</td>
<td>224.8</td>
<td>263.5</td>
</tr>
<tr>
<td>5-heap</td>
<td>274.3</td>
<td>227.9</td>
<td>263.5</td>
</tr>
<tr>
<td>Binomial</td>
<td>274.3</td>
<td>284.1</td>
<td>263.5</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>274.3</td>
<td>320.6</td>
<td>263.5</td>
</tr>
<tr>
<td>Relaxed</td>
<td>274.3</td>
<td>354.9</td>
<td>263.5</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>274.3</td>
<td>147.0</td>
<td>263.5</td>
</tr>
<tr>
<td>2-level 512</td>
<td>274.3</td>
<td>153.7</td>
<td>263.5</td>
</tr>
<tr>
<td>3-level 64</td>
<td>274.3</td>
<td>161.8</td>
<td>263.5</td>
</tr>
<tr>
<td>4-level 24</td>
<td>274.3</td>
<td>166.6</td>
<td>263.5</td>
</tr>
<tr>
<td>5-level 12</td>
<td>274.3</td>
<td>172.0</td>
<td>263.5</td>
</tr>
<tr>
<td>6-level 8</td>
<td>274.3</td>
<td>175.3</td>
<td>263.5</td>
</tr>
<tr>
<td>radix</td>
<td>274.3</td>
<td>202.8</td>
<td>263.5</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>9314.1</td>
<td>4577.5</td>
<td>8823.9</td>
</tr>
<tr>
<td>Parent check</td>
<td>4655.4</td>
<td>2562.9</td>
<td>4427.3</td>
</tr>
<tr>
<td>LLL</td>
<td>352.9</td>
<td>192.1</td>
<td>342.0</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>332.0</td>
<td>197.7</td>
<td>320.7</td>
</tr>
<tr>
<td>d’Esopo-P.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>1496.4</td>
<td>374.8</td>
<td>1540.5</td>
</tr>
<tr>
<td>LLL</td>
<td>308.6</td>
<td>166.1</td>
<td>299.9</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>672.7</td>
<td>267.6</td>
<td>650.1</td>
</tr>
<tr>
<td>LLL</td>
<td>302.2</td>
<td>165.8</td>
<td>292.5</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>1165.3</td>
<td>594.8</td>
<td>1128.0</td>
</tr>
<tr>
<td>Parent check</td>
<td>1000.3</td>
<td>541.5</td>
<td>964.1</td>
</tr>
<tr>
<td>LLL</td>
<td>294.7</td>
<td>162.8</td>
<td>283.6</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>290.4</td>
<td>168.6</td>
<td>279.7</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>276.8</td>
<td>166.6</td>
<td>266.0</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>299.0</td>
<td>132.2</td>
<td>287.1</td>
</tr>
</tbody>
</table>
Table 7.7: Bidirectional methods (A*, balanced) on the full graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th>Method</th>
<th>Dantzig</th>
<th>Pohl</th>
<th>Nicholson</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iter.</td>
<td>CPU</td>
<td>Iter.</td>
</tr>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>79.8</td>
<td>82.7</td>
<td>78.8</td>
</tr>
<tr>
<td>3-heap</td>
<td>79.8</td>
<td>80.7</td>
<td>78.8</td>
</tr>
<tr>
<td>4-heap</td>
<td>79.8</td>
<td>79.7</td>
<td>78.8</td>
</tr>
<tr>
<td>5-heap</td>
<td>79.8</td>
<td>80.1</td>
<td>78.8</td>
</tr>
<tr>
<td>Binomial</td>
<td>79.8</td>
<td>100.0</td>
<td>78.8</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>79.8</td>
<td>109.9</td>
<td>78.8</td>
</tr>
<tr>
<td>Relaxed</td>
<td>79.8</td>
<td>121.6</td>
<td>78.8</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>79.8</td>
<td>54.7</td>
<td>78.8</td>
</tr>
<tr>
<td>2-level 512</td>
<td>79.8</td>
<td>57.6</td>
<td>78.8</td>
</tr>
<tr>
<td>3-level 64</td>
<td>79.8</td>
<td>60.5</td>
<td>78.8</td>
</tr>
<tr>
<td>4-level 24</td>
<td>79.8</td>
<td>62.7</td>
<td>78.8</td>
</tr>
<tr>
<td>5-level 12</td>
<td>79.8</td>
<td>64.3</td>
<td>78.8</td>
</tr>
<tr>
<td>6-level 8</td>
<td>79.8</td>
<td>65.7</td>
<td>78.8</td>
</tr>
<tr>
<td>radix</td>
<td>79.8</td>
<td>73.6</td>
<td>78.8</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>7023.5</td>
<td>3783.4</td>
<td>6722.0</td>
</tr>
<tr>
<td>Parent check</td>
<td>3444.4</td>
<td>2152.5</td>
<td>3314.0</td>
</tr>
<tr>
<td>LLL</td>
<td>143.5</td>
<td>93.2</td>
<td>142.5</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>131.6</td>
<td>97.0</td>
<td>131.0</td>
</tr>
<tr>
<td>d’Esopo-P.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>1356.4</td>
<td>395.7</td>
<td>1407.9</td>
</tr>
<tr>
<td>LLL</td>
<td>134.0</td>
<td>84.6</td>
<td>133.7</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>604.0</td>
<td>277.3</td>
<td>582.6</td>
</tr>
<tr>
<td>LLL</td>
<td>124.1</td>
<td>83.5</td>
<td>123.9</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>296.6</td>
<td>170.5</td>
<td>294.5</td>
</tr>
<tr>
<td>Parent check</td>
<td>268.6</td>
<td>165.8</td>
<td>264.6</td>
</tr>
<tr>
<td>LLL</td>
<td>89.9</td>
<td>59.4</td>
<td>88.9</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>88.3</td>
<td>62.3</td>
<td>87.4</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>80.0</td>
<td>60.2</td>
<td>79.1</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>81.9</td>
<td>47.1</td>
<td>80.9</td>
</tr>
</tbody>
</table>
lower bound for \( d(v, w) \). For any two nodes \( v \) and \( w \) in the middle between \( s \) and \( t \), we may state that \( \delta(v, w) \approx \delta(s, w) - \delta(s, v) \) and \( \delta(v, w) \approx \delta(v, t) - \delta(w, t) \). From these approximations it can readily be derived that for an arc \( (v, w) \) somewhere in the middle of \( s \) and \( t \), the modified arc cost \( c'_{vw} = c_{vw} + h_w - h_v \approx c_{vw} + \delta(v, w) \), no matter whether \( h \) is implemented in a balanced or a symmetric way.

Table 7.8: Bidirectional methods (A*, symmetric) on the full graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LS, heaps</td>
<td>79.4</td>
<td>53.9</td>
<td>79.4</td>
<td>56.1</td>
<td>79.4</td>
<td>79.4</td>
</tr>
<tr>
<td>Binary heap</td>
<td>79.4</td>
<td>77.1</td>
<td>78.1</td>
<td>80.1</td>
<td>82.5</td>
<td>82.5</td>
</tr>
<tr>
<td>3-heap</td>
<td>79.4</td>
<td>77.1</td>
<td>78.1</td>
<td>80.1</td>
<td>82.5</td>
<td>82.5</td>
</tr>
<tr>
<td>4-heap</td>
<td>79.4</td>
<td>77.1</td>
<td>78.1</td>
<td>80.1</td>
<td>82.5</td>
<td>82.5</td>
</tr>
<tr>
<td>5-heap</td>
<td>79.4</td>
<td>77.1</td>
<td>78.1</td>
<td>80.1</td>
<td>82.5</td>
<td>82.5</td>
</tr>
<tr>
<td>Binomial</td>
<td>79.4</td>
<td>77.1</td>
<td>78.1</td>
<td>80.1</td>
<td>82.5</td>
<td>82.5</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>79.4</td>
<td>80.1</td>
<td>78.1</td>
<td>80.1</td>
<td>82.5</td>
<td>82.5</td>
</tr>
<tr>
<td>Relaxed</td>
<td>79.4</td>
<td>80.1</td>
<td>78.1</td>
<td>80.1</td>
<td>82.5</td>
<td>82.5</td>
</tr>
<tr>
<td>LS, buckets</td>
<td>79.4</td>
<td>53.9</td>
<td>78.1</td>
<td>51.4</td>
<td>82.5</td>
<td>82.5</td>
</tr>
<tr>
<td>Dial</td>
<td>79.4</td>
<td>56.1</td>
<td>78.1</td>
<td>53.3</td>
<td>82.5</td>
<td>82.5</td>
</tr>
<tr>
<td>2-level 512</td>
<td>79.4</td>
<td>60.1</td>
<td>78.1</td>
<td>58.6</td>
<td>82.5</td>
<td>82.5</td>
</tr>
<tr>
<td>3-level 64</td>
<td>79.4</td>
<td>60.1</td>
<td>78.1</td>
<td>58.6</td>
<td>82.5</td>
<td>82.5</td>
</tr>
<tr>
<td>4-level 24</td>
<td>79.4</td>
<td>60.1</td>
<td>78.1</td>
<td>58.6</td>
<td>82.5</td>
<td>82.5</td>
</tr>
<tr>
<td>5-level 12</td>
<td>79.4</td>
<td>60.1</td>
<td>78.1</td>
<td>58.6</td>
<td>82.5</td>
<td>82.5</td>
</tr>
<tr>
<td>6-level 8</td>
<td>79.4</td>
<td>60.1</td>
<td>78.1</td>
<td>58.6</td>
<td>82.5</td>
<td>82.5</td>
</tr>
<tr>
<td>radix</td>
<td>79.4</td>
<td>60.1</td>
<td>78.1</td>
<td>60.1</td>
<td>82.5</td>
<td>82.5</td>
</tr>
</tbody>
</table>

In Section 5.3.4 we divide the search that has to be done in the so-called post-phase, based on scalar projections. Table 7.9 shows the results when scalar projections are used to divide the search during the post-phase between forward and backward iterations. Compared to the balanced approach (Table 7.7) a reduction is achieved for all label-setting methods in terms of both the number of iterations and the running times. A reduction is also found for some label-correcting algorithms, especially those that try to emulate a good node ordering based on the label values. Pohl’s alternation rule turns out to be the most efficient and the APPROXIMATE BUCKETS method runs the fastest for each alternation rule that is tested.

7.2.2.2 Bidirectional search using landmarks

Table 7.10 shows the results if the optimization algorithms are implemented in a bidirectional way, using a balanced landmark estimator. For each SSSD-SPP in our testset, 4 active landmarks are chosen, based on our selection criterium, see Section 7.2.1.1.

For the label-setting algorithms, the number of iterations and the running times are roughly halved compared to the unidirectional case, where a Depth First Search from \( t \) is started first (see Table 7.5). When compared to the basic unidirectional methods that use a landmark estimator, the number of iterations is decreased with
Table 7.9: Bidirectional methods (A*, scalar projections) on the full graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th>Method</th>
<th>Dantzig</th>
<th></th>
<th>Pohl</th>
<th></th>
<th>Nicholson</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iter.</td>
<td>CPU</td>
<td>Iter.</td>
<td>CPU</td>
<td>Iter.</td>
<td>CPU</td>
</tr>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>65.9</td>
<td>65.6</td>
<td>65.2</td>
<td>58.8</td>
<td>69.4</td>
<td>68.7</td>
</tr>
<tr>
<td>3-heap</td>
<td>65.9</td>
<td>63.5</td>
<td>65.2</td>
<td>58.1</td>
<td>69.4</td>
<td>67.2</td>
</tr>
<tr>
<td>4-heap</td>
<td>65.9</td>
<td>63.8</td>
<td>65.2</td>
<td>57.8</td>
<td>69.4</td>
<td>67.1</td>
</tr>
<tr>
<td>5-heap</td>
<td>65.9</td>
<td>63.7</td>
<td>65.2</td>
<td>58.5</td>
<td>69.4</td>
<td>66.8</td>
</tr>
<tr>
<td>Binomial</td>
<td>65.9</td>
<td>78.3</td>
<td>65.2</td>
<td>72.0</td>
<td>69.4</td>
<td>89.0</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>65.9</td>
<td>85.7</td>
<td>65.2</td>
<td>77.2</td>
<td>69.4</td>
<td>89.3</td>
</tr>
<tr>
<td>Relaxed</td>
<td>65.9</td>
<td>95.7</td>
<td>65.2</td>
<td>89.0</td>
<td>69.4</td>
<td>113.0</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>65.9</td>
<td>44.6</td>
<td>65.2</td>
<td>41.6</td>
<td>69.4</td>
<td>47.4</td>
</tr>
<tr>
<td>2-level 512</td>
<td>65.9</td>
<td>46.0</td>
<td>65.2</td>
<td>43.5</td>
<td>69.4</td>
<td>49.7</td>
</tr>
<tr>
<td>3-level 64</td>
<td>65.9</td>
<td>48.8</td>
<td>65.2</td>
<td>45.3</td>
<td>69.4</td>
<td>51.8</td>
</tr>
<tr>
<td>4-level 24</td>
<td>65.9</td>
<td>49.6</td>
<td>65.2</td>
<td>46.9</td>
<td>69.4</td>
<td>53.0</td>
</tr>
<tr>
<td>5-level 12</td>
<td>65.9</td>
<td>51.5</td>
<td>65.2</td>
<td>47.6</td>
<td>69.4</td>
<td>54.9</td>
</tr>
<tr>
<td>6-level 8</td>
<td>65.9</td>
<td>51.9</td>
<td>65.2</td>
<td>48.9</td>
<td>69.4</td>
<td>55.5</td>
</tr>
<tr>
<td>radix</td>
<td>65.9</td>
<td>58.1</td>
<td>65.2</td>
<td>53.7</td>
<td>69.4</td>
<td>61.8</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>12097.3</td>
<td>7253.2</td>
<td>12475.1</td>
<td>6864.3</td>
<td>15443.9</td>
<td>8719.5</td>
</tr>
<tr>
<td>Parent check</td>
<td>5515.2</td>
<td>3573.7</td>
<td>5253.1</td>
<td>3237.8</td>
<td>6630.5</td>
<td>4260.3</td>
</tr>
<tr>
<td>LLL</td>
<td>115.2</td>
<td>70.9</td>
<td>114.9</td>
<td>67.6</td>
<td>133.5</td>
<td>78.7</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>105.8</td>
<td>74.4</td>
<td>105.1</td>
<td>68.8</td>
<td>121.9</td>
<td>81.1</td>
</tr>
<tr>
<td>d’Esopo-P.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>1377.2</td>
<td>397.4</td>
<td>1422.4</td>
<td>355.0</td>
<td>1479.7</td>
<td>409.5</td>
</tr>
<tr>
<td>LLL</td>
<td>105.1</td>
<td>65.3</td>
<td>104.8</td>
<td>61.0</td>
<td>119.5</td>
<td>71.2</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>617.2</td>
<td>285.1</td>
<td>596.3</td>
<td>244.6</td>
<td>710.5</td>
<td>307.5</td>
</tr>
<tr>
<td>LLL</td>
<td>96.8</td>
<td>64.1</td>
<td>96.3</td>
<td>60.3</td>
<td>109.5</td>
<td>70.9</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>295.8</td>
<td>171.5</td>
<td>294.6</td>
<td>162.9</td>
<td>338.5</td>
<td>192.4</td>
</tr>
<tr>
<td>Parent check</td>
<td>246.0</td>
<td>149.9</td>
<td>245.4</td>
<td>145.3</td>
<td>285.3</td>
<td>169.5</td>
</tr>
<tr>
<td>LLL</td>
<td>74.3</td>
<td>48.4</td>
<td>73.0</td>
<td>44.4</td>
<td>85.6</td>
<td>53.2</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>72.4</td>
<td>49.1</td>
<td>71.5</td>
<td>46.5</td>
<td>83.6</td>
<td>53.8</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>66.8</td>
<td>46.8</td>
<td>66.2</td>
<td>44.3</td>
<td>70.2</td>
<td>49.6</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>72.5</td>
<td>38.2</td>
<td>71.5</td>
<td>37.7</td>
<td>76.3</td>
<td>40.5</td>
</tr>
</tbody>
</table>
Table 7.10: Bidirectional methods (Landmarks, balanced) on the full graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>5.3</td>
<td>6.3</td>
<td>5.3</td>
<td>5.8</td>
<td>5.0</td>
<td>5.6</td>
</tr>
<tr>
<td>3-heap</td>
<td>5.3</td>
<td>6.4</td>
<td>5.2</td>
<td>5.8</td>
<td>5.0</td>
<td>5.5</td>
</tr>
<tr>
<td>4-heap</td>
<td>5.3</td>
<td>6.1</td>
<td>5.2</td>
<td>5.6</td>
<td>5.0</td>
<td>5.4</td>
</tr>
<tr>
<td>5-heap</td>
<td>5.3</td>
<td>6.5</td>
<td>5.2</td>
<td>5.7</td>
<td>5.0</td>
<td>5.5</td>
</tr>
<tr>
<td>Binomial</td>
<td>5.3</td>
<td>7.2</td>
<td>5.2</td>
<td>6.3</td>
<td>5.0</td>
<td>6.6</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>5.3</td>
<td>7.3</td>
<td>5.2</td>
<td>6.8</td>
<td>5.0</td>
<td>6.6</td>
</tr>
<tr>
<td>Relaxed</td>
<td>5.3</td>
<td>8.3</td>
<td>5.2</td>
<td>7.4</td>
<td>5.0</td>
<td>8.0</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>5.3</td>
<td>4.8</td>
<td>5.2</td>
<td>4.5</td>
<td>5.0</td>
<td>4.4</td>
</tr>
<tr>
<td>2-level 512</td>
<td>5.3</td>
<td>5.3</td>
<td>5.2</td>
<td>4.7</td>
<td>5.0</td>
<td>4.6</td>
</tr>
<tr>
<td>3-level 64</td>
<td>5.3</td>
<td>5.1</td>
<td>5.2</td>
<td>4.8</td>
<td>5.0</td>
<td>4.6</td>
</tr>
<tr>
<td>4-level 24</td>
<td>5.3</td>
<td>5.5</td>
<td>5.2</td>
<td>4.9</td>
<td>5.0</td>
<td>4.8</td>
</tr>
<tr>
<td>5-level 12</td>
<td>5.3</td>
<td>5.3</td>
<td>5.2</td>
<td>4.9</td>
<td>5.0</td>
<td>4.8</td>
</tr>
<tr>
<td>6-level 8</td>
<td>5.3</td>
<td>5.7</td>
<td>5.2</td>
<td>5.0</td>
<td>5.0</td>
<td>4.9</td>
</tr>
<tr>
<td>radix</td>
<td>5.3</td>
<td>5.6</td>
<td>5.2</td>
<td>5.3</td>
<td>5.0</td>
<td>5.1</td>
</tr>
<tr>
<td>BFM</td>
<td>7718.6</td>
<td>4169.8</td>
<td>7459.8</td>
<td>3926.5</td>
<td>10319.1</td>
<td>5833.6</td>
</tr>
<tr>
<td>Parent check</td>
<td>3660.8</td>
<td>2328.9</td>
<td>3552.9</td>
<td>2206.2</td>
<td>4941.8</td>
<td>3250.1</td>
</tr>
<tr>
<td>LLL</td>
<td>106.0</td>
<td>63.3</td>
<td>105.6</td>
<td>60.8</td>
<td>139.0</td>
<td>76.9</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>89.3</td>
<td>68.6</td>
<td>91.5</td>
<td>63.7</td>
<td>117.9</td>
<td>79.2</td>
</tr>
<tr>
<td>d'Esopo-P.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>1358.0</td>
<td>477.0</td>
<td>1410.1</td>
<td>425.7</td>
<td>1506.4</td>
<td>509.5</td>
</tr>
<tr>
<td>LLL</td>
<td>102.4</td>
<td>73.2</td>
<td>102.0</td>
<td>66.3</td>
<td>123.4</td>
<td>77.0</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>608.6</td>
<td>355.0</td>
<td>588.7</td>
<td>308.6</td>
<td>721.6</td>
<td>392.5</td>
</tr>
<tr>
<td>LLL</td>
<td>82.4</td>
<td>70.6</td>
<td>85.5</td>
<td>64.3</td>
<td>100.5</td>
<td>73.9</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>11.2</td>
<td>8.3</td>
<td>12.0</td>
<td>8.1</td>
<td>17.2</td>
<td>11.9</td>
</tr>
<tr>
<td>Parent check</td>
<td>10.5</td>
<td>7.7</td>
<td>11.0</td>
<td>7.8</td>
<td>16.8</td>
<td>12.2</td>
</tr>
<tr>
<td>LLL</td>
<td>5.8</td>
<td>5.0</td>
<td>7.4</td>
<td>5.8</td>
<td>11.4</td>
<td>9.0</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>5.7</td>
<td>4.8</td>
<td>7.4</td>
<td>6.0</td>
<td>11.2</td>
<td>9.2</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>5.28</td>
<td>4.81</td>
<td>7.1</td>
<td>6.2</td>
<td>5.3</td>
<td>4.8</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>6.12</td>
<td>4.93</td>
<td>7.2</td>
<td>5.7</td>
<td>5.1</td>
<td>4.2</td>
</tr>
</tbody>
</table>
an average factor of 86% and the running time is decreased with an average factor of 78%. As we saw earlier a bidirectional approach that uses a balanced Euclidean estimator only results in a speedup of about 40% compared with its unidirectional counterpart. Note that the alteration rule of Pohl does not lead to the overall best running times. The number of iterations varies for label-setting methods if Pohl’s alternation rule is used. For label-setting methods the forward and the backward search meet each other the soonest when Nicholson’s rule is followed.

We implemented the label-setting methods in a bidirectional way, that uses a symmetric landmark estimator as well. The results are shown in Table 7.11. Obviously, using a symmetric landmark estimator cannot compete with a balanced landmark estimator. Even the unidirectional methods that use both the landmark estimator and an initial Depth First Search from \( t \) perform better than the methods that use a symmetric landmark estimator. Only the original unidirectional landmark approach is still outperformed by the bidirectional symmetric landmark approach (in case either Dantzig’s or Pohl’s alternation rule is used).

Table 7.11: Bidirectional methods (Landmarks, symmetric) on the full graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th>Method</th>
<th>Dantzig</th>
<th>Pohl</th>
<th>Nicholson</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iter.</td>
<td>CPU</td>
<td>Iter.</td>
</tr>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>21.2</td>
<td>19.1</td>
<td>27.3</td>
</tr>
<tr>
<td>3-heap</td>
<td>21.7</td>
<td>19.2</td>
<td>28.0</td>
</tr>
<tr>
<td>4-heap</td>
<td>21.4</td>
<td>18.6</td>
<td>27.7</td>
</tr>
<tr>
<td>5-heap</td>
<td>21.4</td>
<td>18.6</td>
<td>28.1</td>
</tr>
<tr>
<td>Binomial</td>
<td>18.3</td>
<td>17.7</td>
<td>24.0</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>19.5</td>
<td>19.9</td>
<td>25.9</td>
</tr>
<tr>
<td>Relaxed</td>
<td>19.2</td>
<td>23.0</td>
<td>23.5</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>19.5</td>
<td>14.0</td>
<td>24.7</td>
</tr>
<tr>
<td>2-level 512</td>
<td>19.5</td>
<td>14.7</td>
<td>24.7</td>
</tr>
<tr>
<td>3-level 64</td>
<td>19.5</td>
<td>14.9</td>
<td>24.7</td>
</tr>
<tr>
<td>4-level 24</td>
<td>19.5</td>
<td>15.0</td>
<td>24.7</td>
</tr>
<tr>
<td>5-level 12</td>
<td>19.5</td>
<td>15.0</td>
<td>24.7</td>
</tr>
<tr>
<td>6-level 8</td>
<td>19.5</td>
<td>15.2</td>
<td>24.7</td>
</tr>
<tr>
<td>radix</td>
<td>19.5</td>
<td>15.3</td>
<td>24.7</td>
</tr>
</tbody>
</table>

Figure 7.5 shows the searched areas for different landmark approaches. As can be seen from this picture, the bidirectional symmetric approach has reached more nodes than the basic unidirectional version. Pohl [157] already mentioned this possible drawback of bidirectional search. As a metaphor he mentioned that it is like two missiles that are independently aimed at each others base in the hope that they will collide somewhere in the middle. Clearly, in a symmetric approach the forward and backward estimator are independent of each other, while in the balanced approach there is a dependency on both the origin as well as the destination. Instead of aiming the heuristics towards the end nodes (the ‘missile bases’), Champeaux et al. [42, 43] proposed to aim at the front nodes of the opposite search. Politowski and Pohl
suggested to choose just one node of the opposite front as target node, and periodically determine another node from that front to be the new target. In the literature these attempts are called ‘wave-shaping’. Instead of just two search trees that are tiny ‘slivers’ in the total search space, it is attempted to shape the search trees towards each other. Although some progress is made in reducing the search space, the time needed for periodically updating the heuristic values outweighs the profits of the reduced search space by far. Kaindl and Kanz [116] declared the missile-metaphor to be wrong and misleading. They stated that the typical problem of the use of symmetric heuristic estimators was that the search spaces of the forward and backward search had a large part in common. The major part of a bidirectional algorithm that uses symmetric heuristic estimators happens after the forward and backward search spaces meet each other for the first time. As we have seen in our experiments this is not the case anymore if either our new bound is used or in case of Euclidean estimators scalar projections are used to divide the work over both searches.

Although Figure 7.5 part (d) shows an example where the search spaces meet each other at a position close to the destination node, one might wonder whether or not this can be called the missile effect. The figure clearly shows search trees that cannot be considered to be just a ‘sliver’. Based on our discussion of landmark estimators for the unidirectional methods (Section 7.2.1.1) we think that, again, the real problem is the vast amount of arcs that has zero reduced arc cost. When a balanced heuristic estimator is used, the number of arcs that has a zero reduced arc costs is considerably less than when a symmetric heuristic estimator is used. Table 7.12 shows the average numbers of arcs with zero modified cost that are encountered during our experiments for each search direction. On the average about 9 times more zero-arc costs are encountered when a symmetric landmark estimator is used.

<table>
<thead>
<tr>
<th>Table 7.12: Average number (in thousands) of arcs with zero modified arc costs.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Estimator</strong></td>
</tr>
<tr>
<td></td>
</tr>
<tr>
<td>Symmetric landmark estimator</td>
</tr>
<tr>
<td>Balanced landmark estimator</td>
</tr>
</tbody>
</table>

Since the use of an initial Depth First Search that only considers arc with zero reduced arc costs turns out to be fruitful for the unidirectional approach, we tried to use this in a bidirectional approach as well. Not only a Depth First Search is grown from $t$, but also from $s$. Hence, a set of destinations $T$ and a set of origin nodes $S$ is determined. Typically, $S \cap T \neq \emptyset$. Each node $v \in S \cap T$ provides an upper bound to $d(s, t)$, since $d(s, v) + d(v, t) = h_t(v) + h_s(v) \geq d(s, t)$. We used the best upper bound found to prune the search in each direction. More important, as soon as the forward search encounters a node $v \in T$ the search in both directions is stopped, since the forward direction alone already provides a solution. In a similar way, the backward search alone has found a solution as soon as a node $v \in S$ gets its final value $d_t^v$. Although we were able to reduce the running times of the symmetric landmark
7.2 Experiments on the full graph

Figure 7.5: The searched area for different landmark approaches. Blue dots denote nodes that are reached by the forward search and green dots denote nodes that are reached either by the backward search or by the initial Depth First Search.
approach by about 50%, it still cannot compete with the balanced approach.

Until now, the main reason for using a balanced approach that is mentioned in the literature, is that as soon as the forward search and the backward search meet each other, only a simple postprocessing step has to be executed in order to determine a shortest path. For a symmetric approach, the amount of work that has to be done after the searches met is considered to be the bottleneck. Although our new bound reduces the running time of the so-called post-phase heavily, we here found another reason to use a balanced approach. If the estimator $h$ leads to a vast amount of arcs that has zero reduced arc costs, a balanced approach is to be preferred over a symmetric approach. Based on the experiments in this chapter, the landmark-based estimator should be applied in a balanced way, while the Euclidean estimator performs better in a symmetric way.

### 7.2.3 Partial shortest path trees

We encountered some real life instances where preprocessing of the graph seems not that useful. We already mentioned that in Emergency Vehicle Routing problems the area that can be reached within a certain amount of time should be determined. Hence, we grow a shortest path tree, until that time bound is reached. Table 7.13 shows the number of iterations and the running time needed to grow a partial shortest path tree, until the time bound of 10, 20, 30 and 40 minutes is reached.

Some of our general observations (Section 7.2.1.2) also hold for the results in Table 7.13. However, there are some differences as well. For the small shortest path tree that is bounded to 10 minutes, the Large Label Last optimization does not lead to a decrease in running time neither for d’Esopo-Pape nor for Pallottino. Furthermore, for each determined shortest path tree, even the basic versions of both algorithms as well as the basic version of SMALL LABEL FIRST outperform the label-setting methods that are heap-based. Among all algorithms the APPROXIMATE BUCKETS method is the fastest. Among all label-setting methods the one where the candidate set $Q$ is implemented using Dial’s bucket data structure is the fastest. Dial’s method and the approximate buckets method are closely related to each other (see Section B.2.3).
### Table 7.13: Growing 100 partial shortest path trees on the full graph. Cumulated running times (milliseconds) and iterations (in thousands).

<table>
<thead>
<tr>
<th>Method</th>
<th>10 minutes</th>
<th>20 minutes</th>
<th>30 minutes</th>
<th>40 minutes</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iter.</td>
<td>CPU</td>
<td>Iter.</td>
<td>CPU</td>
</tr>
<tr>
<td><strong>LS, heaps</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>216.7</td>
<td>119.1</td>
<td>1250.2</td>
<td>678.7</td>
</tr>
<tr>
<td>3-heap</td>
<td>216.7</td>
<td>118.4</td>
<td>1250.2</td>
<td>655.4</td>
</tr>
<tr>
<td>4-heap</td>
<td>216.7</td>
<td>108.4</td>
<td>1250.2</td>
<td>658.4</td>
</tr>
<tr>
<td>5-heap</td>
<td>216.7</td>
<td>109.5</td>
<td>1250.2</td>
<td>665.1</td>
</tr>
<tr>
<td>Binomial</td>
<td>216.7</td>
<td>129.0</td>
<td>1250.2</td>
<td>812.9</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>216.7</td>
<td>136.5</td>
<td>1250.2</td>
<td>860.8</td>
</tr>
<tr>
<td>Relaxed</td>
<td>216.7</td>
<td>158.0</td>
<td>1250.2</td>
<td>1007.4</td>
</tr>
<tr>
<td><strong>LS, buckets</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>216.7</td>
<td>88.8</td>
<td>1250.2</td>
<td>489.6</td>
</tr>
<tr>
<td>2-level 512</td>
<td>216.7</td>
<td>93.2</td>
<td>1250.2</td>
<td>516.7</td>
</tr>
<tr>
<td>3-level 64</td>
<td>216.7</td>
<td>94.5</td>
<td>1250.2</td>
<td>530.1</td>
</tr>
<tr>
<td>4-level 24</td>
<td>216.7</td>
<td>95.5</td>
<td>1250.2</td>
<td>539.5</td>
</tr>
<tr>
<td>5-level 12</td>
<td>216.7</td>
<td>98.0</td>
<td>1250.2</td>
<td>552.2</td>
</tr>
<tr>
<td>6-level 8</td>
<td>216.7</td>
<td>100.0</td>
<td>1250.2</td>
<td>568.8</td>
</tr>
<tr>
<td>radix</td>
<td>216.7</td>
<td>108.2</td>
<td>1250.2</td>
<td>624.8</td>
</tr>
<tr>
<td><strong>BFM</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>414.5</td>
<td>108.7</td>
<td>4264.8</td>
<td>905.8</td>
</tr>
<tr>
<td>Parent check</td>
<td>331.2</td>
<td>103.0</td>
<td>3002.0</td>
<td>772.9</td>
</tr>
<tr>
<td>LLL</td>
<td>236.0</td>
<td>97.9</td>
<td>1443.8</td>
<td>572.4</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>228.9</td>
<td>99.4</td>
<td>1379.2</td>
<td>581.4</td>
</tr>
<tr>
<td><strong>d’Esopo-P.</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>259.5</td>
<td>89.2</td>
<td>1821.7</td>
<td>532.9</td>
</tr>
<tr>
<td>LLL</td>
<td>223.6</td>
<td>94.7</td>
<td>1315.6</td>
<td>539.9</td>
</tr>
<tr>
<td><strong>Pallottino</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>255.7</td>
<td>89.1</td>
<td>1673.0</td>
<td>526.1</td>
</tr>
<tr>
<td>LLL</td>
<td>223.5</td>
<td>95.0</td>
<td>1312.6</td>
<td>539.6</td>
</tr>
<tr>
<td><strong>SLF</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>266.5</td>
<td>91.5</td>
<td>1855.2</td>
<td>567.1</td>
</tr>
<tr>
<td>Parent check</td>
<td>252.1</td>
<td>92.9</td>
<td>1711.9</td>
<td>566.8</td>
</tr>
<tr>
<td>LLL</td>
<td>221.3</td>
<td>90.6</td>
<td>1292.1</td>
<td>508.5</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>220.5</td>
<td>92.8</td>
<td>1284.6</td>
<td>526.3</td>
</tr>
<tr>
<td><strong>Other</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>220.5</td>
<td>89.1</td>
<td>1268.5</td>
<td>506.1</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>216.8</td>
<td>82.7</td>
<td>1250.6</td>
<td>461.9</td>
</tr>
</tbody>
</table>
7.3 Statically preprocessed graph

Once we applied the static preprocessing to the input graph, there are shortcuts added to the graph (to bypass degree 2 nodes). As stated in Table 6.2 we have $|\mathcal{A}| = 22,299,835$, and only 2,657,508 nodes are so-called core nodes. Each non-core cell in the partition contains at most 2,127 nodes. Hence, during each SSSD-SPP at most 31% of the total number of nodes can be visited.

Although the other preprocessing techniques we discuss later in this chapter surely will lead to faster algorithms, the statically preprocessed graph has some nice features. First, the preprocessing time takes only some 20 seconds. Furthermore, if the arc costs have to be updated, the extra work besides updating the arc costs of each arc is that the arc costs of each shortcut has to be updated as well. The shortcuts itself are still valid, since they only bypassed degree 2 nodes. Clearly, this update is very fast. In several real-time environments we consider the static preprocessing to be useful due to the ease of updating and the fact that the running time for an average SSSD-SSP is fast enough for such environments.

We run our test set on this graph. Table 7.14 shows the results of the unidirectional methods. For the landmark estimator we choose four active landmarks for each individual SSSD-SPP using our criterium as described in Section 6.3.3.2. We also implemented the variant that uses a Depth First Search from the destination node $t$ in the reverse graph that contains only nodes with modified arc cost zero, see Section 7.2.1 for more details.

Compared to Tables 7.4 and 7.5 the running time is reduced by an average factor of 58%. For the label-setting methods the number of iterations is reduced by an average factor of 70%. Since the number of core-nodes is about 31%, this observation was to be expected. Our general observations (Section 7.2.1.2) also hold for the results on the statically preprocessed graph.

Table 7.15 shows the results for bidirectional methods that use no estimators. Pohl’s alternation rule turns out to be the most efficient. Compared to Table 7.6 the number of iterations is reduced by an average factor of 69% and the running time is reduced by an average factor of 57%.

We applied the Euclidean estimator to the bidirectional implementations in both a balanced (Table 7.16) and a symmetric way (Table 7.17). The running times of the symmetric approach only slightly improve the ones of the balanced approach. Furthermore, the expected speedups compared to the experiments on the full graph are achieved, both in terms of the number of iterations as well as the running times. Again, Pohl’s alternation rule turns out to be the most efficient. Among all tests on the statically preprocessed graph that use a Euclidean estimator, the scalar projection methods performs the best (see Table 7.18).
Table 7.14: Unidirectional methods on the graph after static preprocessing. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th>Method</th>
<th>No estimator</th>
<th>A⁺</th>
<th>Landmarks</th>
<th>Landm. dfs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iter.</td>
<td>CPU</td>
<td>Iter.</td>
<td>CPU</td>
</tr>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>132.9</td>
<td>134.7</td>
<td>42.8</td>
<td>51.8</td>
</tr>
<tr>
<td>3-heap</td>
<td>132.9</td>
<td>131.2</td>
<td>42.8</td>
<td>51.1</td>
</tr>
<tr>
<td>4-heap</td>
<td>132.9</td>
<td>131.6</td>
<td>42.8</td>
<td>50.9</td>
</tr>
<tr>
<td>5-heap</td>
<td>132.9</td>
<td>131.9</td>
<td>42.8</td>
<td>51.2</td>
</tr>
<tr>
<td>Binomial</td>
<td>132.9</td>
<td>157.6</td>
<td>42.8</td>
<td>60.0</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>132.9</td>
<td>173.3</td>
<td>42.8</td>
<td>65.1</td>
</tr>
<tr>
<td>Relaxed</td>
<td>132.9</td>
<td>189.4</td>
<td>42.8</td>
<td>71.3</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>132.9</td>
<td>96.7</td>
<td>42.8</td>
<td>39.4</td>
</tr>
<tr>
<td>2-level 512</td>
<td>132.9</td>
<td>100.9</td>
<td>42.8</td>
<td>40.6</td>
</tr>
<tr>
<td>3-level 64</td>
<td>132.9</td>
<td>104.2</td>
<td>42.8</td>
<td>42.2</td>
</tr>
<tr>
<td>4-level 24</td>
<td>132.9</td>
<td>106.3</td>
<td>42.8</td>
<td>42.8</td>
</tr>
<tr>
<td>5-level 12</td>
<td>132.9</td>
<td>109.2</td>
<td>42.8</td>
<td>43.9</td>
</tr>
<tr>
<td>6-level 8</td>
<td>132.9</td>
<td>111.2</td>
<td>42.8</td>
<td>44.4</td>
</tr>
<tr>
<td>radix</td>
<td>132.9</td>
<td>122.4</td>
<td>42.8</td>
<td>47.9</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>3530.9</td>
<td>2592.3</td>
<td>2377.3</td>
<td>1985.5</td>
</tr>
<tr>
<td>Parent check</td>
<td>2028.1</td>
<td>1623.1</td>
<td>1363.6</td>
<td>1228.2</td>
</tr>
<tr>
<td>LLL</td>
<td>176.4</td>
<td>111.4</td>
<td>66.5</td>
<td>52.4</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>163.5</td>
<td>112.9</td>
<td>60.5</td>
<td>52.4</td>
</tr>
<tr>
<td>d’Esopo-P.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>1248.9</td>
<td>360.2</td>
<td>1134.6</td>
<td>385.8</td>
</tr>
<tr>
<td>LLL</td>
<td>159.8</td>
<td>99.7</td>
<td>65.1</td>
<td>44.9</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>436.3</td>
<td>217.5</td>
<td>398.0</td>
<td>235.9</td>
</tr>
<tr>
<td>LLL</td>
<td>156.2</td>
<td>99.7</td>
<td>56.5</td>
<td>43.5</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>972.2</td>
<td>672.7</td>
<td>395.7</td>
<td>310.5</td>
</tr>
<tr>
<td>Parent check</td>
<td>810.1</td>
<td>591.6</td>
<td>341.0</td>
<td>283.5</td>
</tr>
<tr>
<td>LLL</td>
<td>153.9</td>
<td>99.3</td>
<td>51.2</td>
<td>42.2</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>150.7</td>
<td>101.8</td>
<td>50.1</td>
<td>43.9</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>134.5</td>
<td>103.3</td>
<td>43.4</td>
<td>41.5</td>
</tr>
<tr>
<td>Approximate b.</td>
<td>148.7</td>
<td>92.7</td>
<td>48.3</td>
<td>36.4</td>
</tr>
</tbody>
</table>
Table 7.15: Bidirectional methods (no estimator) on the statically preprocessed graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>83.8</td>
<td>91.7</td>
<td>80.2</td>
<td>79.3</td>
<td>89.5</td>
<td>98.1</td>
</tr>
<tr>
<td>3-heap</td>
<td>83.8</td>
<td>90.0</td>
<td>80.2</td>
<td>78.6</td>
<td>89.5</td>
<td>96.6</td>
</tr>
<tr>
<td>4-heap</td>
<td>83.8</td>
<td>90.0</td>
<td>80.2</td>
<td>81.1</td>
<td>89.5</td>
<td>96.7</td>
</tr>
<tr>
<td>5-heap</td>
<td>83.8</td>
<td>90.9</td>
<td>80.2</td>
<td>82.2</td>
<td>89.5</td>
<td>96.7</td>
</tr>
<tr>
<td>Binomial</td>
<td>83.8</td>
<td>106.6</td>
<td>80.2</td>
<td>96.2</td>
<td>89.5</td>
<td>124.1</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>83.8</td>
<td>117.3</td>
<td>80.2</td>
<td>103.6</td>
<td>89.5</td>
<td>126.7</td>
</tr>
<tr>
<td>Relaxed</td>
<td>83.8</td>
<td>126.8</td>
<td>80.2</td>
<td>112.0</td>
<td>89.5</td>
<td>151.8</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>83.8</td>
<td>66.6</td>
<td>80.2</td>
<td>59.6</td>
<td>89.5</td>
<td>74.5</td>
</tr>
<tr>
<td>2-level 512</td>
<td>83.8</td>
<td>69.5</td>
<td>80.2</td>
<td>61.9</td>
<td>89.5</td>
<td>76.1</td>
</tr>
<tr>
<td>3-level 64</td>
<td>83.8</td>
<td>72.9</td>
<td>80.2</td>
<td>63.3</td>
<td>89.5</td>
<td>78.5</td>
</tr>
<tr>
<td>4-level 24</td>
<td>83.8</td>
<td>73.2</td>
<td>80.2</td>
<td>65.4</td>
<td>89.5</td>
<td>79.8</td>
</tr>
<tr>
<td>5-level 12</td>
<td>83.8</td>
<td>74.7</td>
<td>80.2</td>
<td>69.3</td>
<td>89.5</td>
<td>81.9</td>
</tr>
<tr>
<td>6-level 8</td>
<td>83.8</td>
<td>78.2</td>
<td>80.2</td>
<td>68.4</td>
<td>89.5</td>
<td>82.8</td>
</tr>
<tr>
<td>radix</td>
<td>83.8</td>
<td>83.8</td>
<td>80.2</td>
<td>75.3</td>
<td>89.5</td>
<td>90.6</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>1493.1</td>
<td>1198.3</td>
<td>1420.9</td>
<td>1084.1</td>
<td>1928.6</td>
<td>1553.7</td>
</tr>
<tr>
<td>Parent check</td>
<td>839.9</td>
<td>738.9</td>
<td>810.8</td>
<td>679.6</td>
<td>1068.9</td>
<td>929.7</td>
</tr>
<tr>
<td>LLL</td>
<td>108.3</td>
<td>80.9</td>
<td>104.0</td>
<td>71.2</td>
<td>124.3</td>
<td>87.1</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>100.8</td>
<td>82.0</td>
<td>96.9</td>
<td>71.6</td>
<td>116.3</td>
<td>88.0</td>
</tr>
<tr>
<td>d’Esopo-P.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>578.4</td>
<td>208.1</td>
<td>623.6</td>
<td>200.1</td>
<td>703.7</td>
<td>233.4</td>
</tr>
<tr>
<td>LLL</td>
<td>98.4</td>
<td>71.6</td>
<td>94.5</td>
<td>64.5</td>
<td>113.0</td>
<td>79.1</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>237.7</td>
<td>137.8</td>
<td>232.9</td>
<td>122.9</td>
<td>275.0</td>
<td>150.3</td>
</tr>
<tr>
<td>LLL</td>
<td>96.4</td>
<td>71.4</td>
<td>92.6</td>
<td>64.5</td>
<td>111.5</td>
<td>78.9</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>452.3</td>
<td>348.9</td>
<td>438.2</td>
<td>323.6</td>
<td>584.7</td>
<td>439.3</td>
</tr>
<tr>
<td>Parent check</td>
<td>373.3</td>
<td>304.6</td>
<td>363.7</td>
<td>281.9</td>
<td>481.6</td>
<td>380.8</td>
</tr>
<tr>
<td>LLL</td>
<td>95.6</td>
<td>72.7</td>
<td>91.6</td>
<td>63.7</td>
<td>110.4</td>
<td>78.2</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>93.6</td>
<td>74.6</td>
<td>89.6</td>
<td>65.1</td>
<td>108.3</td>
<td>80.2</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>84.3</td>
<td>74.0</td>
<td>80.8</td>
<td>66.3</td>
<td>90.1</td>
<td>78.8</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>93.1</td>
<td>64.0</td>
<td>89.4</td>
<td>57.9</td>
<td>99.5</td>
<td>68.8</td>
</tr>
</tbody>
</table>
Table 7.16: Bidirectional methods (A*, balanced) on the statically preprocessed graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th>Method</th>
<th>Dantzig</th>
<th></th>
<th>Pohl</th>
<th></th>
<th>Nicholson</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iter.</td>
<td>CPU</td>
<td>Iter.</td>
<td>CPU</td>
<td>Iter.</td>
<td>CPU</td>
</tr>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>25.1</td>
<td>33.4</td>
<td>24.7</td>
<td>30.6</td>
<td>26.8</td>
<td>35.8</td>
</tr>
<tr>
<td>3-heap</td>
<td>25.1</td>
<td>32.9</td>
<td>24.7</td>
<td>30.1</td>
<td>26.8</td>
<td>35.1</td>
</tr>
<tr>
<td>4-heap</td>
<td>25.1</td>
<td>32.7</td>
<td>24.7</td>
<td>30.2</td>
<td>26.8</td>
<td>35.1</td>
</tr>
<tr>
<td>5-heap</td>
<td>25.1</td>
<td>32.8</td>
<td>24.7</td>
<td>30.2</td>
<td>26.8</td>
<td>35.1</td>
</tr>
<tr>
<td>Binomial</td>
<td>25.1</td>
<td>38.8</td>
<td>24.7</td>
<td>35.6</td>
<td>26.8</td>
<td>44.0</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>25.1</td>
<td>42.0</td>
<td>24.7</td>
<td>38.1</td>
<td>26.8</td>
<td>44.7</td>
</tr>
<tr>
<td>Relaxed</td>
<td>25.1</td>
<td>45.4</td>
<td>24.7</td>
<td>41.7</td>
<td>26.8</td>
<td>52.8</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>25.1</td>
<td>25.7</td>
<td>24.7</td>
<td>24.2</td>
<td>26.8</td>
<td>27.8</td>
</tr>
<tr>
<td>2-level 512</td>
<td>25.1</td>
<td>26.8</td>
<td>24.7</td>
<td>24.9</td>
<td>26.8</td>
<td>28.8</td>
</tr>
<tr>
<td>3-level 64</td>
<td>25.1</td>
<td>27.4</td>
<td>24.7</td>
<td>25.6</td>
<td>26.8</td>
<td>29.5</td>
</tr>
<tr>
<td>4-level 24</td>
<td>25.1</td>
<td>28.1</td>
<td>24.7</td>
<td>26.0</td>
<td>26.8</td>
<td>30.0</td>
</tr>
<tr>
<td>5-level 12</td>
<td>25.1</td>
<td>28.4</td>
<td>24.7</td>
<td>26.5</td>
<td>26.8</td>
<td>30.6</td>
</tr>
<tr>
<td>6-level 8</td>
<td>25.1</td>
<td>29.0</td>
<td>24.7</td>
<td>27.0</td>
<td>26.8</td>
<td>31.2</td>
</tr>
<tr>
<td>radix</td>
<td>25.1</td>
<td>31.2</td>
<td>24.7</td>
<td>29.0</td>
<td>26.8</td>
<td>33.6</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>676.3</td>
<td>587.0</td>
<td>650.3</td>
<td>548.0</td>
<td>994.6</td>
<td>877.7</td>
</tr>
<tr>
<td>Parent check</td>
<td>380.7</td>
<td>369.0</td>
<td>369.1</td>
<td>344.6</td>
<td>540.6</td>
<td>522.4</td>
</tr>
<tr>
<td>LLL</td>
<td>39.6</td>
<td>35.3</td>
<td>39.1</td>
<td>32.3</td>
<td>44.1</td>
<td>35.7</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>35.8</td>
<td>35.4</td>
<td>35.3</td>
<td>32.3</td>
<td>39.9</td>
<td>35.7</td>
</tr>
<tr>
<td>d’Esopo-P.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>364.2</td>
<td>157.3</td>
<td>394.3</td>
<td>150.6</td>
<td>492.7</td>
<td>191.3</td>
</tr>
<tr>
<td>LLL</td>
<td>35.9</td>
<td>29.4</td>
<td>35.7</td>
<td>27.4</td>
<td>40.1</td>
<td>30.4</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>157.5</td>
<td>106.5</td>
<td>155.7</td>
<td>98.4</td>
<td>196.7</td>
<td>125.8</td>
</tr>
<tr>
<td>LLL</td>
<td>32.7</td>
<td>28.9</td>
<td>32.2</td>
<td>26.8</td>
<td>36.4</td>
<td>29.8</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>149.9</td>
<td>127.3</td>
<td>147.6</td>
<td>122.6</td>
<td>186.5</td>
<td>153.6</td>
</tr>
<tr>
<td>Parent check</td>
<td>125.2</td>
<td>113.9</td>
<td>123.0</td>
<td>108.6</td>
<td>155.2</td>
<td>135.8</td>
</tr>
<tr>
<td>LLL</td>
<td>30.5</td>
<td>28.5</td>
<td>30.0</td>
<td>26.3</td>
<td>34.3</td>
<td>29.3</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>29.7</td>
<td>29.3</td>
<td>29.3</td>
<td>27.0</td>
<td>33.5</td>
<td>30.2</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>25.0</td>
<td>27.3</td>
<td>24.6</td>
<td>25.6</td>
<td>26.7</td>
<td>29.1</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>25.7</td>
<td>23.3</td>
<td>25.4</td>
<td>22.0</td>
<td>27.5</td>
<td>24.8</td>
</tr>
</tbody>
</table>
Table 7.17: Bidirectional methods ($A^*$, symmetric) on the statically preprocessed graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>24.7</td>
<td>31.8</td>
<td>24.2</td>
<td>29.4</td>
<td>25.9</td>
<td>33.2</td>
</tr>
<tr>
<td>3-heap</td>
<td>24.7</td>
<td>31.3</td>
<td>24.2</td>
<td>29.1</td>
<td>25.9</td>
<td>32.7</td>
</tr>
<tr>
<td>4-heap</td>
<td>24.7</td>
<td>31.2</td>
<td>24.2</td>
<td>29.0</td>
<td>25.9</td>
<td>32.7</td>
</tr>
<tr>
<td>5-heap</td>
<td>24.7</td>
<td>31.4</td>
<td>24.2</td>
<td>29.2</td>
<td>25.9</td>
<td>32.8</td>
</tr>
<tr>
<td>Binomial</td>
<td>24.7</td>
<td>36.5</td>
<td>24.2</td>
<td>33.8</td>
<td>25.9</td>
<td>40.0</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>24.7</td>
<td>39.3</td>
<td>24.2</td>
<td>36.2</td>
<td>25.9</td>
<td>41.0</td>
</tr>
<tr>
<td>Relaxed</td>
<td>24.7</td>
<td>42.3</td>
<td>24.2</td>
<td>39.5</td>
<td>25.9</td>
<td>47.9</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>24.7</td>
<td>24.5</td>
<td>24.2</td>
<td>23.0</td>
<td>25.9</td>
<td>26.0</td>
</tr>
<tr>
<td>2-level 512</td>
<td>24.7</td>
<td>25.3</td>
<td>24.2</td>
<td>23.8</td>
<td>25.9</td>
<td>26.7</td>
</tr>
<tr>
<td>3-level 64</td>
<td>24.7</td>
<td>26.0</td>
<td>24.2</td>
<td>24.4</td>
<td>25.9</td>
<td>27.5</td>
</tr>
<tr>
<td>4-level 24</td>
<td>24.7</td>
<td>26.6</td>
<td>24.2</td>
<td>24.9</td>
<td>25.9</td>
<td>27.9</td>
</tr>
<tr>
<td>5-level 12</td>
<td>24.7</td>
<td>26.9</td>
<td>24.2</td>
<td>25.4</td>
<td>25.9</td>
<td>28.4</td>
</tr>
<tr>
<td>6-level 8</td>
<td>24.7</td>
<td>27.4</td>
<td>24.2</td>
<td>25.8</td>
<td>25.9</td>
<td>28.9</td>
</tr>
<tr>
<td>radix</td>
<td>24.7</td>
<td>29.4</td>
<td>24.2</td>
<td>27.6</td>
<td>25.9</td>
<td>30.9</td>
</tr>
</tbody>
</table>

The final test on the statically preprocessed graph is based on bidirectional methods that use a balanced landmark estimator. As seen in Section 7.2.2.2 using the landmark estimator in a symmetric way cannot compete with the balanced approach. As observed earlier, the number of iterations varies for label-setting methods if Pohl’s alternation rule is used. For label-setting methods Nicholson’s alternation rule results in a minimum number of iterations.
Table 7.18: Bidirectional methods (A*, scalar projections) on the statically preprocessed graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th>Method</th>
<th>Dantzig</th>
<th>Pohl</th>
<th>Nicholson</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iter.</td>
<td>CPU</td>
<td>Iter.</td>
</tr>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>20.7</td>
<td>26.6</td>
<td>20.3</td>
</tr>
<tr>
<td>3-heap</td>
<td>20.7</td>
<td>26.1</td>
<td>20.3</td>
</tr>
<tr>
<td>4-heap</td>
<td>20.7</td>
<td>26.1</td>
<td>20.3</td>
</tr>
<tr>
<td>5-heap</td>
<td>20.7</td>
<td>26.3</td>
<td>20.3</td>
</tr>
<tr>
<td>Binomial</td>
<td>20.7</td>
<td>30.7</td>
<td>20.3</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>20.7</td>
<td>32.9</td>
<td>20.3</td>
</tr>
<tr>
<td>Relaxed</td>
<td>20.7</td>
<td>35.5</td>
<td>20.3</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>20.7</td>
<td>20.4</td>
<td>20.3</td>
</tr>
<tr>
<td>2-level 512</td>
<td>20.7</td>
<td>21.1</td>
<td>20.3</td>
</tr>
<tr>
<td>3-level 64</td>
<td>20.7</td>
<td>21.8</td>
<td>20.3</td>
</tr>
<tr>
<td>4-level 24</td>
<td>20.7</td>
<td>22.1</td>
<td>20.3</td>
</tr>
<tr>
<td>5-level 12</td>
<td>20.7</td>
<td>22.6</td>
<td>20.3</td>
</tr>
<tr>
<td>6-level 8</td>
<td>20.7</td>
<td>22.9</td>
<td>20.3</td>
</tr>
<tr>
<td>radix</td>
<td>20.7</td>
<td>24.6</td>
<td>20.3</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>1261.9</td>
<td>1139.5</td>
<td>1209.4</td>
</tr>
<tr>
<td>Parent check</td>
<td>593.1</td>
<td>593.4</td>
<td>579.4</td>
</tr>
<tr>
<td>LLL</td>
<td>33.0</td>
<td>27.9</td>
<td>32.4</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>29.9</td>
<td>28.0</td>
<td>29.4</td>
</tr>
<tr>
<td>d’Esopo-P.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>380.9</td>
<td>162.4</td>
<td>408.8</td>
</tr>
<tr>
<td>LLL</td>
<td>29.7</td>
<td>23.4</td>
<td>29.6</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>162.3</td>
<td>109.6</td>
<td>163.1</td>
</tr>
<tr>
<td>LLL</td>
<td>27.2</td>
<td>23.0</td>
<td>26.9</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>164.6</td>
<td>139.4</td>
<td>164.5</td>
</tr>
<tr>
<td>Parent check</td>
<td>142.9</td>
<td>128.3</td>
<td>136.2</td>
</tr>
<tr>
<td>LLL</td>
<td>25.1</td>
<td>22.3</td>
<td>24.8</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>24.6</td>
<td>22.9</td>
<td>24.2</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>21.0</td>
<td>21.5</td>
<td>20.6</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>23.3</td>
<td>18.2</td>
<td>22.9</td>
</tr>
</tbody>
</table>
### Table 7.19: Bidirectional methods (landmarks, balanced) on the statically preprocessed graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>1.70</td>
<td>2.56</td>
<td>1.69</td>
<td>2.42</td>
<td>1.60</td>
<td>2.28</td>
</tr>
<tr>
<td>3-heap</td>
<td>1.70</td>
<td>2.51</td>
<td>1.70</td>
<td>2.40</td>
<td>1.60</td>
<td>2.25</td>
</tr>
<tr>
<td>4-heap</td>
<td>1.70</td>
<td>2.49</td>
<td>1.71</td>
<td>2.40</td>
<td>1.60</td>
<td>2.24</td>
</tr>
<tr>
<td>5-heap</td>
<td>1.70</td>
<td>2.49</td>
<td>1.69</td>
<td>2.39</td>
<td>1.60</td>
<td>2.25</td>
</tr>
<tr>
<td>Binomial</td>
<td>1.70</td>
<td>2.73</td>
<td>1.70</td>
<td>2.61</td>
<td>1.60</td>
<td>2.57</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1.70</td>
<td>2.90</td>
<td>1.71</td>
<td>2.77</td>
<td>1.60</td>
<td>2.59</td>
</tr>
<tr>
<td>Relaxed</td>
<td>1.70</td>
<td>3.09</td>
<td>1.70</td>
<td>2.98</td>
<td>1.60</td>
<td>3.06</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>1.70</td>
<td>2.11</td>
<td>1.72</td>
<td>2.06</td>
<td>1.60</td>
<td>1.93</td>
</tr>
<tr>
<td>2-level 512</td>
<td>1.70</td>
<td>2.16</td>
<td>1.71</td>
<td>2.12</td>
<td>1.60</td>
<td>1.98</td>
</tr>
<tr>
<td>3-level 64</td>
<td>1.70</td>
<td>2.19</td>
<td>1.69</td>
<td>2.11</td>
<td>1.60</td>
<td>1.99</td>
</tr>
<tr>
<td>4-level 24</td>
<td>1.70</td>
<td>2.21</td>
<td>1.71</td>
<td>2.16</td>
<td>1.60</td>
<td>2.02</td>
</tr>
<tr>
<td>5-level 12</td>
<td>1.70</td>
<td>2.24</td>
<td>1.71</td>
<td>2.18</td>
<td>1.60</td>
<td>2.04</td>
</tr>
<tr>
<td>6-level 8</td>
<td>1.70</td>
<td>2.26</td>
<td>1.71</td>
<td>2.21</td>
<td>1.60</td>
<td>2.06</td>
</tr>
<tr>
<td>radix</td>
<td>1.70</td>
<td>2.35</td>
<td>1.70</td>
<td>2.27</td>
<td>1.60</td>
<td>2.14</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>628.46</td>
<td>570.35</td>
<td>606.54</td>
<td>531.94</td>
<td>979.35</td>
<td>886.19</td>
</tr>
<tr>
<td>Parent check</td>
<td>344.04</td>
<td>350.72</td>
<td>337.85</td>
<td>336.95</td>
<td>522.66</td>
<td>529.72</td>
</tr>
<tr>
<td>LLL</td>
<td>19.76</td>
<td>16.32</td>
<td>19.71</td>
<td>15.88</td>
<td>26.51</td>
<td>21.43</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>17.03</td>
<td>16.87</td>
<td>17.21</td>
<td>16.31</td>
<td>25.85</td>
<td>24.35</td>
</tr>
<tr>
<td>d’Esopo-P.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>353.34</td>
<td>173.33</td>
<td>386.96</td>
<td>165.85</td>
<td>493.19</td>
<td>214.04</td>
</tr>
<tr>
<td>LLL</td>
<td>22.53</td>
<td>18.71</td>
<td>23.55</td>
<td>18.12</td>
<td>28.61</td>
<td>23.04</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>155.96</td>
<td>123.97</td>
<td>153.24</td>
<td>114.36</td>
<td>198.65</td>
<td>147.70</td>
</tr>
<tr>
<td>LLL</td>
<td>17.79</td>
<td>17.53</td>
<td>17.75</td>
<td>16.82</td>
<td>23.75</td>
<td>22.16</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>4.36</td>
<td>4.09</td>
<td>4.62</td>
<td>4.26</td>
<td>6.95</td>
<td>6.42</td>
</tr>
<tr>
<td>Parent check</td>
<td>3.78</td>
<td>3.78</td>
<td>3.87</td>
<td>3.78</td>
<td>5.82</td>
<td>5.76</td>
</tr>
<tr>
<td>LLL</td>
<td>2.05</td>
<td>2.23</td>
<td>2.16</td>
<td>2.27</td>
<td>3.24</td>
<td>3.48</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>2.12</td>
<td>2.37</td>
<td>2.13</td>
<td>2.31</td>
<td>3.07</td>
<td>3.39</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>1.72</td>
<td>2.20</td>
<td>1.71</td>
<td>2.14</td>
<td>1.70</td>
<td>2.14</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>1.68</td>
<td>1.97</td>
<td>2.14</td>
<td>2.41</td>
<td>1.62</td>
<td>1.86</td>
</tr>
</tbody>
</table>
7.4 Dynamically preprocessed graph

We applied a dynamic preprocessing on our input graph, based on graph contraction (see Section 6.3.1). We conduct a node reduction with contraction parameters \( c = 2.5 \) and \( h = 50 \), followed by the arc reduction procedure. From Table 6.3 it can be read that the resulting graph contains 153,359 core nodes and a total of 21,593,444 arcs. Since the number of nodes in each cell of the graph partitioning contains at most 26,369 nodes, for any SSSD-SPP, the number of nodes that can be visited is bounded by \( 153,359 + 2 \times 26,369 = 206,277 \). Hence, we might expect a reduction of about 98% in the number of iterations, and hopefully in the running times.

Table 7.20 shows the cumulated running times and iterations for each optimization algorithm, when unidirectional search is applied. The landmark estimator selects four active landmarks for each SSSD-SPP using our selection criterium. The landmark estimator is implemented in its basic form and combined with an initial Depth First Search (see Section 7.2.1.1).

The results follow our general observations as stated in Section 7.2.1.2. Indeed the number of iterations is decreased by 98% on the average for label-setting methods compared to the similar experiments on the full graph. The running times are decreased by 97% if no estimator or the Euclidean estimator is used. The running times are decreased by 96% if the landmark estimator is used in its basic form and by 95% if an initial Depth First Search is performed. The averages mentioned here, are based on the label-setting methods only.

Table 7.21 shows the cumulated running times and iterations for each optimization algorithm. Here, no heuristic estimator is involved. We tested three different alternation rules between the forward and backward search (Dantzig, Pohl and Nicholson). The so-called cardinality comparison rule of Pohl turns out to be the most efficient in this experiment, followed by the turn based rule of Dantzig.

The running times of the algorithms decreases with by 31%–55%. Only Bellman-Ford-Moore (basic and with parent-check improvement) and Small Label First (basic and with parent-check) have a decrease in running time of 50% or more. The running time of the label-setting algorithms decreases by 37% on the average. This is as expected, since especially the label-correcting methods where a node is selected several times as a pivot node before it gets its final label value, gains extra from the decrease in the search space.

Table 7.22 shows the results for each optimization algorithm if the Euclidean (A*) heuristic is used in a balanced way. Pohl’s alternation rule turns out to be superior over the other ones. Compared to the similar experiments on the full graph the number of iterations is reduced by an average factor of 98% and the running times by an average factor of 97%.

Table 7.23 shows the results for each optimization algorithm if the Euclidean (A*) heuristic is used in a symmetric way. Compared to Table 7.22 where the estimator is implemented in a balanced way, the running times only slightly improve by about
Table 7.20: Unidirectional methods on the dynamically preprocessed graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th>Method</th>
<th>No estimator</th>
<th>A*</th>
<th>Landmarks</th>
<th>Landm. dfs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iter.</td>
<td>CPU</td>
<td>Iter.</td>
<td>CPU</td>
</tr>
<tr>
<td><strong>LS, heaps</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>7.90</td>
<td>9.60</td>
<td>2.58</td>
<td>3.75</td>
</tr>
<tr>
<td>3-heap</td>
<td>7.90</td>
<td>9.46</td>
<td>2.58</td>
<td>3.67</td>
</tr>
<tr>
<td>4-heap</td>
<td>7.90</td>
<td>9.33</td>
<td>2.58</td>
<td>3.65</td>
</tr>
<tr>
<td>5-heap</td>
<td>7.90</td>
<td>9.36</td>
<td>2.58</td>
<td>3.67</td>
</tr>
<tr>
<td>Binomial</td>
<td>7.90</td>
<td>10.91</td>
<td>2.58</td>
<td>4.19</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>7.90</td>
<td>11.65</td>
<td>2.58</td>
<td>4.47</td>
</tr>
<tr>
<td>Relaxed</td>
<td>7.90</td>
<td>12.52</td>
<td>2.58</td>
<td>4.77</td>
</tr>
<tr>
<td><strong>LS, buckets</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>7.90</td>
<td>8.08</td>
<td>2.58</td>
<td>3.19</td>
</tr>
<tr>
<td>2-level 512</td>
<td>7.90</td>
<td>8.08</td>
<td>2.58</td>
<td>3.21</td>
</tr>
<tr>
<td>3-level 64</td>
<td>7.90</td>
<td>8.32</td>
<td>2.58</td>
<td>3.26</td>
</tr>
<tr>
<td>4-level 24</td>
<td>7.90</td>
<td>8.43</td>
<td>2.58</td>
<td>3.46</td>
</tr>
<tr>
<td>5-level 12</td>
<td>7.90</td>
<td>8.66</td>
<td>2.58</td>
<td>3.38</td>
</tr>
<tr>
<td>6-level 8</td>
<td>7.90</td>
<td>8.80</td>
<td>2.58</td>
<td>3.48</td>
</tr>
<tr>
<td>radix</td>
<td>7.90</td>
<td>9.39</td>
<td>2.58</td>
<td>3.67</td>
</tr>
<tr>
<td><strong>BFM</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>50.93</td>
<td>44.78</td>
<td>40.97</td>
<td>44.28</td>
</tr>
<tr>
<td>Parent check</td>
<td>35.98</td>
<td>33.76</td>
<td>29.35</td>
<td>33.90</td>
</tr>
<tr>
<td>LLL</td>
<td>9.86</td>
<td>8.60</td>
<td>4.70</td>
<td>4.52</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>9.48</td>
<td>8.55</td>
<td>4.39</td>
<td>4.57</td>
</tr>
<tr>
<td><strong>d’Esopo-P.</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>18.13</td>
<td>11.27</td>
<td>17.67</td>
<td>13.52</td>
</tr>
<tr>
<td>LLL</td>
<td>9.40</td>
<td>8.16</td>
<td>4.47</td>
<td>4.39</td>
</tr>
<tr>
<td><strong>Pallottino</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>15.11</td>
<td>10.65</td>
<td>14.74</td>
<td>12.72</td>
</tr>
<tr>
<td>LLL</td>
<td>9.36</td>
<td>8.15</td>
<td>4.32</td>
<td>4.36</td>
</tr>
<tr>
<td><strong>SLF</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>34.71</td>
<td>29.68</td>
<td>18.48</td>
<td>19.31</td>
</tr>
<tr>
<td>Parent check</td>
<td>24.71</td>
<td>22.57</td>
<td>14.22</td>
<td>15.87</td>
</tr>
<tr>
<td>LLL</td>
<td>9.12</td>
<td>7.82</td>
<td>3.71</td>
<td>3.73</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>8.90</td>
<td>7.89</td>
<td>3.55</td>
<td>3.76</td>
</tr>
<tr>
<td><strong>Other</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>7.91</td>
<td>9.55</td>
<td>2.59</td>
<td>3.63</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>7.93</td>
<td>7.52</td>
<td>2.60</td>
<td>2.96</td>
</tr>
</tbody>
</table>
Table 7.21: Bidirectional methods (no estimator) on the dynamically preprocessed graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th>Method</th>
<th>Dantzig</th>
<th>Pohl</th>
<th>Nicholson</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iter.</td>
<td>CPU</td>
<td>Iter.</td>
</tr>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>5.10</td>
<td>6.73</td>
<td>4.86</td>
</tr>
<tr>
<td>3-heap</td>
<td>5.10</td>
<td>6.51</td>
<td>4.86</td>
</tr>
<tr>
<td>4-heap</td>
<td>5.10</td>
<td>6.46</td>
<td>4.86</td>
</tr>
<tr>
<td>5-heap</td>
<td>5.10</td>
<td>6.45</td>
<td>4.86</td>
</tr>
<tr>
<td>Binomial</td>
<td>5.10</td>
<td>7.45</td>
<td>4.86</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>5.10</td>
<td>8.08</td>
<td>4.86</td>
</tr>
<tr>
<td>Relaxed</td>
<td>5.10</td>
<td>8.51</td>
<td>4.86</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>5.10</td>
<td>5.55</td>
<td>4.86</td>
</tr>
<tr>
<td>2-level 512</td>
<td>5.10</td>
<td>5.63</td>
<td>4.86</td>
</tr>
<tr>
<td>3-level 64</td>
<td>5.10</td>
<td>5.74</td>
<td>4.86</td>
</tr>
<tr>
<td>4-level 24</td>
<td>5.10</td>
<td>5.87</td>
<td>4.86</td>
</tr>
<tr>
<td>5-level 12</td>
<td>5.10</td>
<td>6.00</td>
<td>4.86</td>
</tr>
<tr>
<td>6-level 8</td>
<td>5.10</td>
<td>6.12</td>
<td>4.86</td>
</tr>
<tr>
<td>radix</td>
<td>5.10</td>
<td>6.50</td>
<td>4.86</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>23.46</td>
<td>22.74</td>
<td>22.69</td>
</tr>
<tr>
<td>Parent check</td>
<td>17.11</td>
<td>17.93</td>
<td>16.61</td>
</tr>
<tr>
<td>LLL</td>
<td>6.00</td>
<td>5.71</td>
<td>5.74</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>5.75</td>
<td>5.78</td>
<td>5.50</td>
</tr>
<tr>
<td>d’Esopo-P.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>9.17</td>
<td>6.93</td>
<td>9.01</td>
</tr>
<tr>
<td>LLL</td>
<td>5.61</td>
<td>5.35</td>
<td>5.37</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>8.02</td>
<td>6.58</td>
<td>7.77</td>
</tr>
<tr>
<td>LLL</td>
<td>5.59</td>
<td>5.36</td>
<td>5.35</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>16.46</td>
<td>15.57</td>
<td>15.92</td>
</tr>
<tr>
<td>Parent check</td>
<td>12.43</td>
<td>12.58</td>
<td>12.03</td>
</tr>
<tr>
<td>LLL</td>
<td>5.70</td>
<td>5.50</td>
<td>5.45</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>5.55</td>
<td>5.59</td>
<td>5.31</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>5.06</td>
<td>6.74</td>
<td>4.85</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>5.09</td>
<td>5.51</td>
<td>4.87</td>
</tr>
</tbody>
</table>
Table 7.22: Bidirectional methods (A*, balanced) on the dynamically preprocessed graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>1.62</td>
<td>2.66</td>
<td>1.61</td>
<td>2.48</td>
<td>1.72</td>
<td>2.80</td>
</tr>
<tr>
<td>3-heap</td>
<td>1.62</td>
<td>2.60</td>
<td>1.61</td>
<td>2.43</td>
<td>1.72</td>
<td>2.75</td>
</tr>
<tr>
<td>4-heap</td>
<td>1.62</td>
<td>2.56</td>
<td>1.61</td>
<td>2.42</td>
<td>1.72</td>
<td>2.70</td>
</tr>
<tr>
<td>5-heap</td>
<td>1.62</td>
<td>2.55</td>
<td>1.61</td>
<td>2.42</td>
<td>1.72</td>
<td>2.70</td>
</tr>
<tr>
<td>Binomial</td>
<td>1.62</td>
<td>2.91</td>
<td>1.61</td>
<td>2.76</td>
<td>1.72</td>
<td>3.22</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1.62</td>
<td>3.12</td>
<td>1.61</td>
<td>2.94</td>
<td>1.72</td>
<td>3.29</td>
</tr>
<tr>
<td>Relaxed</td>
<td>1.62</td>
<td>3.29</td>
<td>1.61</td>
<td>3.14</td>
<td>1.72</td>
<td>3.77</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>1.62</td>
<td>2.31</td>
<td>1.61</td>
<td>2.23</td>
<td>1.72</td>
<td>2.43</td>
</tr>
<tr>
<td>2-level 512</td>
<td>1.62</td>
<td>2.28</td>
<td>1.61</td>
<td>2.19</td>
<td>1.72</td>
<td>2.42</td>
</tr>
<tr>
<td>3-level 64</td>
<td>1.62</td>
<td>2.32</td>
<td>1.61</td>
<td>2.22</td>
<td>1.72</td>
<td>2.45</td>
</tr>
<tr>
<td>4-level 24</td>
<td>1.62</td>
<td>2.34</td>
<td>1.61</td>
<td>2.26</td>
<td>1.72</td>
<td>2.49</td>
</tr>
<tr>
<td>5-level 12</td>
<td>1.62</td>
<td>2.39</td>
<td>1.61</td>
<td>2.29</td>
<td>1.72</td>
<td>2.54</td>
</tr>
<tr>
<td>6-level 8</td>
<td>1.62</td>
<td>2.43</td>
<td>1.61</td>
<td>2.32</td>
<td>1.72</td>
<td>2.58</td>
</tr>
<tr>
<td>radix</td>
<td>1.62</td>
<td>2.55</td>
<td>1.61</td>
<td>2.44</td>
<td>1.72</td>
<td>2.70</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>12.67</td>
<td>14.34</td>
<td>12.35</td>
<td>13.21</td>
<td>14.33</td>
<td>15.50</td>
</tr>
<tr>
<td>Parent check</td>
<td>9.48</td>
<td>11.71</td>
<td>9.26</td>
<td>10.79</td>
<td>10.67</td>
<td>12.64</td>
</tr>
<tr>
<td>LLL</td>
<td>2.78</td>
<td>3.03</td>
<td>2.73</td>
<td>2.89</td>
<td>2.94</td>
<td>3.09</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>2.60</td>
<td>3.13</td>
<td>2.55</td>
<td>2.92</td>
<td>2.74</td>
<td>3.10</td>
</tr>
<tr>
<td>d’Esopo-P.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>6.99</td>
<td>6.33</td>
<td>6.94</td>
<td>5.93</td>
<td>7.60</td>
<td>6.60</td>
</tr>
<tr>
<td>LLL</td>
<td>2.47</td>
<td>2.72</td>
<td>2.45</td>
<td>2.64</td>
<td>2.62</td>
<td>2.83</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>6.22</td>
<td>6.13</td>
<td>6.07</td>
<td>5.60</td>
<td>6.69</td>
<td>6.35</td>
</tr>
<tr>
<td>LLL</td>
<td>2.42</td>
<td>2.71</td>
<td>2.39</td>
<td>2.63</td>
<td>2.56</td>
<td>2.81</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>7.02</td>
<td>7.84</td>
<td>6.88</td>
<td>7.39</td>
<td>7.77</td>
<td>8.33</td>
</tr>
<tr>
<td>Parent check</td>
<td>5.69</td>
<td>6.86</td>
<td>5.55</td>
<td>6.40</td>
<td>6.31</td>
<td>7.28</td>
</tr>
<tr>
<td>LLL</td>
<td>2.30</td>
<td>2.63</td>
<td>2.27</td>
<td>2.50</td>
<td>2.44</td>
<td>2.69</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>2.20</td>
<td>2.65</td>
<td>2.16</td>
<td>2.52</td>
<td>2.34</td>
<td>2.72</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>1.61</td>
<td>2.68</td>
<td>1.59</td>
<td>2.58</td>
<td>1.69</td>
<td>2.81</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>1.61</td>
<td>2.08</td>
<td>1.59</td>
<td>2.00</td>
<td>1.69</td>
<td>2.14</td>
</tr>
</tbody>
</table>
7.4 Dynamically preprocessed graph

1%. Only Dial’s implementation gains 4.8%. We already discussed why one might expect that the number of iterations is more or less equal for the balanced and the symmetric approach (see Section 7.2.2.1).

Table 7.23: Bidirectional methods (A*, symmetric) on the dynamically preprocessed graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>1.58</td>
<td>2.59</td>
<td>1.56</td>
<td>2.44</td>
<td>1.64</td>
<td>2.65</td>
</tr>
<tr>
<td>3-heap</td>
<td>1.58</td>
<td>2.52</td>
<td>1.56</td>
<td>2.40</td>
<td>1.64</td>
<td>2.60</td>
</tr>
<tr>
<td>4-heap</td>
<td>1.58</td>
<td>2.50</td>
<td>1.56</td>
<td>2.39</td>
<td>1.64</td>
<td>2.58</td>
</tr>
<tr>
<td>5-heap</td>
<td>1.58</td>
<td>2.50</td>
<td>1.56</td>
<td>2.39</td>
<td>1.64</td>
<td>2.59</td>
</tr>
<tr>
<td>Binomial</td>
<td>1.58</td>
<td>2.83</td>
<td>1.56</td>
<td>2.72</td>
<td>1.64</td>
<td>3.02</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1.58</td>
<td>3.07</td>
<td>1.56</td>
<td>2.88</td>
<td>1.64</td>
<td>3.12</td>
</tr>
<tr>
<td>Relaxed</td>
<td>1.58</td>
<td>3.29</td>
<td>1.56</td>
<td>3.11</td>
<td>1.64</td>
<td>3.53</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>1.58</td>
<td>2.19</td>
<td>1.56</td>
<td>2.13</td>
<td>1.64</td>
<td>2.26</td>
</tr>
<tr>
<td>2-level 512</td>
<td>1.58</td>
<td>2.24</td>
<td>1.56</td>
<td>2.17</td>
<td>1.64</td>
<td>2.31</td>
</tr>
<tr>
<td>3-level 64</td>
<td>1.58</td>
<td>2.27</td>
<td>1.56</td>
<td>2.18</td>
<td>1.64</td>
<td>2.35</td>
</tr>
<tr>
<td>4-level 24</td>
<td>1.58</td>
<td>2.33</td>
<td>1.56</td>
<td>2.22</td>
<td>1.64</td>
<td>2.41</td>
</tr>
<tr>
<td>5-level 12</td>
<td>1.58</td>
<td>2.35</td>
<td>1.56</td>
<td>2.26</td>
<td>1.64</td>
<td>2.42</td>
</tr>
<tr>
<td>6-level 8</td>
<td>1.58</td>
<td>2.38</td>
<td>1.56</td>
<td>2.29</td>
<td>1.64</td>
<td>2.46</td>
</tr>
<tr>
<td>radix</td>
<td>1.58</td>
<td>2.49</td>
<td>1.56</td>
<td>2.40</td>
<td>1.64</td>
<td>2.56</td>
</tr>
</tbody>
</table>

If we use scalar projections to divide the search between forward and backward iterations after the first meeting node is found, the symmetric approach (using a Euclidean estimator) outperforms the balanced approach in case of the label-setting methods by about 19%. The so-called potential invariant label-correcting methods shows an increase in running time. This is probably due to the relatively late iteration that the first meeting node is found. Table 7.24 shows the running times for the scalar projection symmetric approach.

Table 7.25 shows the results of running the different optimization methods in a bidirectional way, using a balanced landmark estimator. Similar to the experiments on the full graph and on the statically preprocessed graph, Nicholson’s alternation rule results in the fastest running times when a balanced landmark estimator is used. By comparing Table 7.25 with the results of the unidirectional methods of Table 7.20 we can see that implementing the landmark estimator in a bidirectionally balanced way reduces the running times of the label-setting methods by a factor of about 68%, compared with their unidirectional counterparts that use the landmark estimator in its basic form. Compared to the unidirectional implementations that first run a Depth First Search, the running time is reduced by an average factor of 33%.
Table 7.24: Bidirectional methods (A*, scalar projections) on the dynamically preprocessed graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>1.38</td>
<td>2.21</td>
<td>1.33</td>
<td>2.01</td>
<td>1.42</td>
<td>2.26</td>
</tr>
<tr>
<td>3-heap</td>
<td>1.38</td>
<td>2.17</td>
<td>1.33</td>
<td>1.97</td>
<td>1.42</td>
<td>2.21</td>
</tr>
<tr>
<td>4-heap</td>
<td>1.38</td>
<td>2.15</td>
<td>1.33</td>
<td>1.97</td>
<td>1.42</td>
<td>2.20</td>
</tr>
<tr>
<td>5-heap</td>
<td>1.38</td>
<td>2.14</td>
<td>1.33</td>
<td>1.97</td>
<td>1.42</td>
<td>2.20</td>
</tr>
<tr>
<td>Binomial</td>
<td>1.38</td>
<td>2.54</td>
<td>1.33</td>
<td>2.26</td>
<td>1.42</td>
<td>2.63</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1.38</td>
<td>2.71</td>
<td>1.33</td>
<td>2.38</td>
<td>1.42</td>
<td>2.66</td>
</tr>
<tr>
<td>Relaxed</td>
<td>1.38</td>
<td>2.80</td>
<td>1.33</td>
<td>2.55</td>
<td>1.42</td>
<td>3.11</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>1.38</td>
<td>1.87</td>
<td>1.33</td>
<td>1.75</td>
<td>1.42</td>
<td>1.92</td>
</tr>
<tr>
<td>2-level 512</td>
<td>1.38</td>
<td>1.90</td>
<td>1.33</td>
<td>1.76</td>
<td>1.42</td>
<td>1.96</td>
</tr>
<tr>
<td>3-level 64</td>
<td>1.38</td>
<td>1.93</td>
<td>1.33</td>
<td>1.79</td>
<td>1.42</td>
<td>1.98</td>
</tr>
<tr>
<td>4-level 24</td>
<td>1.38</td>
<td>1.95</td>
<td>1.33</td>
<td>1.83</td>
<td>1.42</td>
<td>2.02</td>
</tr>
<tr>
<td>5-level 12</td>
<td>1.38</td>
<td>2.00</td>
<td>1.33</td>
<td>1.85</td>
<td>1.42</td>
<td>2.06</td>
</tr>
<tr>
<td>6-level 8</td>
<td>1.38</td>
<td>2.03</td>
<td>1.33</td>
<td>1.88</td>
<td>1.42</td>
<td>2.09</td>
</tr>
<tr>
<td>radix</td>
<td>1.38</td>
<td>2.13</td>
<td>1.33</td>
<td>1.99</td>
<td>1.42</td>
<td>2.19</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>15.67</td>
<td>17.47</td>
<td>14.99</td>
<td>15.61</td>
<td>16.96</td>
<td>18.11</td>
</tr>
<tr>
<td>Parent check</td>
<td>11.08</td>
<td>13.76</td>
<td>10.62</td>
<td>12.06</td>
<td>12.24</td>
<td>14.27</td>
</tr>
<tr>
<td>LLL</td>
<td>2.39</td>
<td>2.53</td>
<td>2.35</td>
<td>2.42</td>
<td>2.51</td>
<td>2.57</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>2.19</td>
<td>2.49</td>
<td>2.16</td>
<td>2.42</td>
<td>2.28</td>
<td>2.53</td>
</tr>
<tr>
<td>d’Esopo-P.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>7.06</td>
<td>6.40</td>
<td>6.99</td>
<td>5.96</td>
<td>7.72</td>
<td>6.64</td>
</tr>
<tr>
<td>LLL</td>
<td>2.05</td>
<td>2.21</td>
<td>2.03</td>
<td>2.16</td>
<td>2.15</td>
<td>2.27</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>6.24</td>
<td>6.12</td>
<td>6.13</td>
<td>5.71</td>
<td>6.72</td>
<td>6.31</td>
</tr>
<tr>
<td>LLL</td>
<td>2.00</td>
<td>2.20</td>
<td>1.98</td>
<td>2.15</td>
<td>2.09</td>
<td>2.26</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>7.23</td>
<td>7.92</td>
<td>7.08</td>
<td>7.46</td>
<td>7.88</td>
<td>8.31</td>
</tr>
<tr>
<td>Parent check</td>
<td>5.84</td>
<td>6.83</td>
<td>5.74</td>
<td>6.42</td>
<td>6.25</td>
<td>7.04</td>
</tr>
<tr>
<td>LLL</td>
<td>1.90</td>
<td>2.09</td>
<td>1.86</td>
<td>2.01</td>
<td>2.02</td>
<td>2.18</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>1.82</td>
<td>2.12</td>
<td>1.78</td>
<td>2.03</td>
<td>1.94</td>
<td>2.22</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>1.38</td>
<td>2.06</td>
<td>1.34</td>
<td>1.95</td>
<td>1.43</td>
<td>2.11</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>1.41</td>
<td>1.73</td>
<td>1.35</td>
<td>1.61</td>
<td>1.45</td>
<td>1.73</td>
</tr>
</tbody>
</table>
Table 7.25: Bidirectional methods (landmarks, balanced) on the dynamically preprocessed graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th>Method</th>
<th>Dantzig Iter.</th>
<th>CPU</th>
<th>Pohl Iter.</th>
<th>CPU</th>
<th>Nicholson Iter.</th>
<th>CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>0.18</td>
<td>0.38</td>
<td>0.18</td>
<td>0.34</td>
<td>0.17</td>
<td>0.31</td>
</tr>
<tr>
<td>3-heap</td>
<td>0.18</td>
<td>0.37</td>
<td>0.18</td>
<td>0.33</td>
<td>0.17</td>
<td>0.31</td>
</tr>
<tr>
<td>4-heap</td>
<td>0.18</td>
<td>0.36</td>
<td>0.18</td>
<td>0.33</td>
<td>0.17</td>
<td>0.31</td>
</tr>
<tr>
<td>5-heap</td>
<td>0.18</td>
<td>0.36</td>
<td>0.18</td>
<td>0.33</td>
<td>0.17</td>
<td>0.31</td>
</tr>
<tr>
<td>Binomial</td>
<td>0.18</td>
<td>0.40</td>
<td>0.18</td>
<td>0.36</td>
<td>0.17</td>
<td>0.34</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>0.18</td>
<td>0.59</td>
<td>0.18</td>
<td>0.38</td>
<td>0.17</td>
<td>0.35</td>
</tr>
<tr>
<td>Relaxed</td>
<td>0.18</td>
<td>0.43</td>
<td>0.18</td>
<td>0.40</td>
<td>0.17</td>
<td>0.39</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>0.18</td>
<td>0.34</td>
<td>0.18</td>
<td>0.32</td>
<td>0.17</td>
<td>0.29</td>
</tr>
<tr>
<td>2-level 512</td>
<td>0.18</td>
<td>0.34</td>
<td>0.18</td>
<td>0.31</td>
<td>0.17</td>
<td>0.29</td>
</tr>
<tr>
<td>3-level 64</td>
<td>0.18</td>
<td>0.34</td>
<td>0.18</td>
<td>0.31</td>
<td>0.17</td>
<td>0.30</td>
</tr>
<tr>
<td>4-level 24</td>
<td>0.18</td>
<td>0.34</td>
<td>0.18</td>
<td>0.31</td>
<td>0.17</td>
<td>0.29</td>
</tr>
<tr>
<td>5-level 12</td>
<td>0.18</td>
<td>0.35</td>
<td>0.18</td>
<td>0.32</td>
<td>0.17</td>
<td>0.30</td>
</tr>
<tr>
<td>6-level 8</td>
<td>0.18</td>
<td>0.35</td>
<td>0.18</td>
<td>0.32</td>
<td>0.17</td>
<td>0.30</td>
</tr>
<tr>
<td>radix</td>
<td>0.18</td>
<td>0.36</td>
<td>0.18</td>
<td>0.33</td>
<td>0.17</td>
<td>0.31</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>11.41</td>
<td>14.65</td>
<td>11.11</td>
<td>13.54</td>
<td>13.63</td>
<td>16.51</td>
</tr>
<tr>
<td>Parent check</td>
<td>8.62</td>
<td>12.19</td>
<td>8.44</td>
<td>11.53</td>
<td>10.05</td>
<td>14.07</td>
</tr>
<tr>
<td>LLL</td>
<td>1.99</td>
<td>2.49</td>
<td>2.08</td>
<td>2.51</td>
<td>2.55</td>
<td>2.91</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>1.84</td>
<td>2.53</td>
<td>1.90</td>
<td>2.51</td>
<td>2.26</td>
<td>2.87</td>
</tr>
<tr>
<td>d’Esopo-P.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>6.98</td>
<td>7.76</td>
<td>6.94</td>
<td>7.15</td>
<td>7.66</td>
<td>8.11</td>
</tr>
<tr>
<td>LLL</td>
<td>1.95</td>
<td>2.51</td>
<td>1.94</td>
<td>2.44</td>
<td>2.31</td>
<td>2.82</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>6.21</td>
<td>7.49</td>
<td>6.09</td>
<td>6.86</td>
<td>6.79</td>
<td>7.80</td>
</tr>
<tr>
<td>LLL</td>
<td>1.86</td>
<td>2.49</td>
<td>1.87</td>
<td>2.43</td>
<td>2.21</td>
<td>2.79</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>0.47</td>
<td>0.62</td>
<td>0.45</td>
<td>0.57</td>
<td>0.57</td>
<td>0.73</td>
</tr>
<tr>
<td>Parent check</td>
<td>0.41</td>
<td>0.59</td>
<td>0.39</td>
<td>0.54</td>
<td>0.52</td>
<td>0.70</td>
</tr>
<tr>
<td>LLL</td>
<td>0.31</td>
<td>0.44</td>
<td>0.28</td>
<td>0.39</td>
<td>0.36</td>
<td>0.50</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>0.30</td>
<td>0.45</td>
<td>0.27</td>
<td>0.40</td>
<td>0.34</td>
<td>0.51</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>0.19</td>
<td>0.41</td>
<td>0.19</td>
<td>0.39</td>
<td>0.18</td>
<td>0.37</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>0.19</td>
<td>0.33</td>
<td>0.18</td>
<td>0.30</td>
<td>0.18</td>
<td>0.29</td>
</tr>
</tbody>
</table>
7.5 Hybrid preprocessing of graphs

The hybrid preprocessing of graphs that we consider is based on a graph partitioning that contains 329 cells. Each cell contains at most 32,768 nodes. The number of border nodes is 17,480. As described in Section 6.5 for each cell, shortcuts between any pair of its border nodes are added. These shortcuts contain only nodes that are part of the cell. In this way, 958,843 shortcuts are added to the graph. Hence, our graph has now the following characteristics \( |V| = 8,678,011 \) and \( |A| = 20,531,184 \).

For any SSSD-SPP at most \( 2 \times 32,768 + 17,480 = 83,016 \) nodes are searched, since only the core nodes (here, the border nodes) and at most two cells are involved.

Table 7.26 shows the cumulated running times and iterations for each optimization algorithm, when unidirectional search is applied. Our general observations hold for the numbers in this table as well. Applying an initial Depth First Search to the landmark estimator again stabilizes the number of iterations for label-setting methods an decreases the running times with 10%–87%.

Table 7.27 shows the cumulated running times if the optimization methods are implemented in a bidirectional way, for three different alternation rules. Although these numbers confirm our earlier observation that the Pohl’s alternation rule outperform the other two, there is something interesting in this table. Bidirectional search methods (without an estimator) seem to have a worse running time and need a larger number of iterations to complete, compared to unidirectional search methods (see Table 7.26). Only some bidirectional label-correcting methods perform slightly better than their unidirectional counterparts. But for label-setting methods the increase in running time, compared to their unidirectional counterparts is considerable: 46% on the average.

It is not difficult to explain why the bidirectional implementations are inferior to the unidirectional ones on our hybrid preprocessed graph. Since the average cell-size (number of nodes in the cell) is higher than the number of core-nodes it might happen that in bidirectional search more nodes are scanned, especially in the cell that contains the destination node. This effect is visualized in Figure 7.6. Clearly one can see that the searched area in the cell of destination node \( t \) is larger in the bidirectional case.

Neither the use of adding a balanced Euclidean estimator (see Table 7.28), nor adding a symmetric Euclidean estimator (see Table 7.29) overcomes the problem that is depicted in Figure 7.6. Using a balanced approach increases the running time by an average factor of 26% for label-setting methods. For label-setting methods that use a symmetric Euclidean estimator the running time still increases by an average factor of 21%.

Unfortunately, even the bidirectional methods that are based on scalar projections (Table 7.30) and the bidirectional methods that use a landmark-based estimator in a balanced way (Table 7.31) cannot compete with their unidirectional counterparts.
Table 7.26: Unidirectional methods on the hybrid preprocessed graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th>Method</th>
<th>No estimator</th>
<th>A*</th>
<th>Landmarks</th>
<th>Landm. dfs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iter. CPU</td>
<td>Iter. CPU</td>
<td>Iter. CPU</td>
<td>Iter. CPU</td>
</tr>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>5.20 4.75</td>
<td>3.69 3.35</td>
<td>1.60 1.46</td>
<td>1.07 1.03</td>
</tr>
<tr>
<td>3-heap</td>
<td>5.20 4.74</td>
<td>3.69 3.34</td>
<td>1.58 1.38</td>
<td>1.07 1.02</td>
</tr>
<tr>
<td>4-heap</td>
<td>5.20 4.66</td>
<td>3.69 3.33</td>
<td>1.57 1.37</td>
<td>1.07 1.03</td>
</tr>
<tr>
<td>5-heap</td>
<td>5.20 4.66</td>
<td>3.69 3.36</td>
<td>1.56 1.38</td>
<td>1.07 1.04</td>
</tr>
<tr>
<td>Binomial</td>
<td>5.20 5.48</td>
<td>3.69 3.97</td>
<td>1.47 1.39</td>
<td>1.07 1.11</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>5.20 5.70</td>
<td>3.69 4.08</td>
<td>1.47 1.48</td>
<td>1.07 1.17</td>
</tr>
<tr>
<td>Relaxed</td>
<td>5.20 6.47</td>
<td>3.69 4.67</td>
<td>1.52 1.78</td>
<td>1.07 1.31</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>5.20 4.12</td>
<td>3.69 2.82</td>
<td>1.44 1.08</td>
<td>1.07 0.91</td>
</tr>
<tr>
<td>2-level 512</td>
<td>5.20 4.01</td>
<td>3.69 2.83</td>
<td>1.44 1.10</td>
<td>1.07 0.92</td>
</tr>
<tr>
<td>3-level 64</td>
<td>5.20 4.06</td>
<td>3.69 2.88</td>
<td>1.44 1.11</td>
<td>1.07 0.94</td>
</tr>
<tr>
<td>4-level 24</td>
<td>5.20 4.10</td>
<td>3.69 2.91</td>
<td>1.45 1.11</td>
<td>1.07 0.93</td>
</tr>
<tr>
<td>5-level 12</td>
<td>5.20 4.21</td>
<td>3.69 2.97</td>
<td>1.44 1.12</td>
<td>1.07 0.93</td>
</tr>
<tr>
<td>6-level 8</td>
<td>5.20 4.24</td>
<td>3.69 3.00</td>
<td>1.45 1.13</td>
<td>1.07 0.93</td>
</tr>
<tr>
<td>radix</td>
<td>5.20 4.52</td>
<td>3.69 3.17</td>
<td>1.44 1.13</td>
<td>1.07 0.94</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>104.04 58.38</td>
<td>86.02 61.19</td>
<td>62.48 55.83</td>
<td>10.06 3.52</td>
</tr>
<tr>
<td>Parent check</td>
<td>63.13 39.62</td>
<td>52.37 41.47</td>
<td>38.29 37.79</td>
<td>6.27 2.59</td>
</tr>
<tr>
<td>LLL</td>
<td>27.95 9.10</td>
<td>27.38 9.37</td>
<td>21.71 10.30</td>
<td>5.06 2.15</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>21.48 8.45</td>
<td>20.84 8.75</td>
<td>16.36 9.75</td>
<td>3.73 1.93</td>
</tr>
<tr>
<td>d’Esopo-P.</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>100.98 38.07</td>
<td>88.57 43.21</td>
<td>66.22 41.61</td>
<td>6.31 2.21</td>
</tr>
<tr>
<td>LLL</td>
<td>54.64 11.48</td>
<td>57.58 14.11</td>
<td>40.61 15.19</td>
<td>4.36 1.80</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>34.67 18.15</td>
<td>31.13 20.89</td>
<td>25.09 21.00</td>
<td>3.02 1.53</td>
</tr>
<tr>
<td>LLL</td>
<td>20.66 7.30</td>
<td>20.03 7.82</td>
<td>15.96 8.87</td>
<td>2.43 1.39</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>34.98 24.73</td>
<td>24.55 22.13</td>
<td>8.33 7.72</td>
<td>2.42 1.34</td>
</tr>
<tr>
<td>Parent check</td>
<td>30.07 22.86</td>
<td>21.90 21.24</td>
<td>7.73 7.44</td>
<td>2.25 1.33</td>
</tr>
<tr>
<td>LLL</td>
<td>14.06 6.19</td>
<td>12.11 5.36</td>
<td>4.68 3.54</td>
<td>1.96 1.21</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>13.21 6.26</td>
<td>11.44 5.63</td>
<td>4.59 3.67</td>
<td>1.89 1.21</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>5.28 5.36</td>
<td>3.76 3.27</td>
<td>2.14 1.68</td>
<td>1.24 1.10</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>5.60 3.73</td>
<td>4.02 2.58</td>
<td>1.53 1.09</td>
<td>1.11 0.90</td>
</tr>
</tbody>
</table>
Table 7.27: Bidirectional methods (no estimator) on the hybrid preprocessed graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th>Method</th>
<th>Dantzig Iter. CPU</th>
<th>Pohl Iter. CPU</th>
<th>Nicholson Iter. CPU</th>
</tr>
</thead>
<tbody>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>6.26 5.40</td>
<td>6.14 4.88</td>
<td>6.41 5.60</td>
</tr>
<tr>
<td>3-heap</td>
<td>6.26 5.36</td>
<td>6.14 4.84</td>
<td>6.41 5.55</td>
</tr>
<tr>
<td>4-heap</td>
<td>6.26 5.34</td>
<td>6.14 4.84</td>
<td>6.41 5.58</td>
</tr>
<tr>
<td>5-heap</td>
<td>6.26 5.43</td>
<td>6.14 4.86</td>
<td>6.41 5.55</td>
</tr>
<tr>
<td>Binomial</td>
<td>6.26 6.36</td>
<td>6.14 5.76</td>
<td>6.41 7.02</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>6.26 6.83</td>
<td>6.14 5.99</td>
<td>6.41 6.82</td>
</tr>
<tr>
<td>Relaxed</td>
<td>6.26 7.49</td>
<td>6.14 6.83</td>
<td>6.41 8.73</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>6.26 4.58</td>
<td>6.14 4.18</td>
<td>6.41 4.77</td>
</tr>
<tr>
<td>2-level 512</td>
<td>6.26 4.60</td>
<td>6.14 4.16</td>
<td>6.41 4.78</td>
</tr>
<tr>
<td>3-level 64</td>
<td>6.26 4.64</td>
<td>6.14 4.20</td>
<td>6.41 4.83</td>
</tr>
<tr>
<td>4-level 24</td>
<td>6.26 4.71</td>
<td>6.14 4.25</td>
<td>6.41 4.92</td>
</tr>
<tr>
<td>5-level 12</td>
<td>6.26 4.81</td>
<td>6.14 4.33</td>
<td>6.41 5.05</td>
</tr>
<tr>
<td>6-level 8</td>
<td>6.26 4.84</td>
<td>6.14 4.39</td>
<td>6.41 5.06</td>
</tr>
<tr>
<td>radix</td>
<td>6.26 5.17</td>
<td>6.14 4.68</td>
<td>6.41 5.32</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>91.16 42.37</td>
<td>96.79 35.19</td>
<td>101.75 39.88</td>
</tr>
<tr>
<td>Parent check</td>
<td>57.60 30.77</td>
<td>60.32 25.51</td>
<td>62.27 29.25</td>
</tr>
<tr>
<td>LLL</td>
<td>36.88 12.67</td>
<td>36.30 11.72</td>
<td>36.61 12.28</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>28.39 11.96</td>
<td>28.10 10.95</td>
<td>28.34 11.24</td>
</tr>
<tr>
<td>d’Esopo-P.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>126.40 34.50</td>
<td>130.96 30.87</td>
<td>131.45 34.92</td>
</tr>
<tr>
<td>LLL</td>
<td>84.72 18.45</td>
<td>81.31 16.48</td>
<td>84.42 18.30</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>39.05 16.12</td>
<td>40.13 13.99</td>
<td>40.49 15.91</td>
</tr>
<tr>
<td>LLL</td>
<td>28.04 9.85</td>
<td>27.32 8.98</td>
<td>27.88 9.63</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>29.12 17.34</td>
<td>29.96 14.43</td>
<td>30.57 15.79</td>
</tr>
<tr>
<td>Parent check</td>
<td>25.93 16.69</td>
<td>26.62 13.85</td>
<td>27.01 15.77</td>
</tr>
<tr>
<td>LLL</td>
<td>17.21 8.12</td>
<td>16.85 7.27</td>
<td>17.17 7.74</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>16.24 8.22</td>
<td>15.83 7.36</td>
<td>16.14 7.76</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>6.44 6.11</td>
<td>6.34 5.78</td>
<td>6.45 6.27</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>6.84 4.68</td>
<td>6.75 4.47</td>
<td>6.86 4.78</td>
</tr>
</tbody>
</table>
Table 7.28: Bidirectional methods (A*, balanced) on the hybrid preprocessed graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th>Method</th>
<th>Dantzig</th>
<th>Pohl</th>
<th>Nicholson</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iter.</td>
<td>CPU</td>
<td>Iter.</td>
</tr>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>5.04</td>
<td>4.77</td>
<td>4.84</td>
</tr>
<tr>
<td>3-heap</td>
<td>5.04</td>
<td>4.68</td>
<td>4.84</td>
</tr>
<tr>
<td>4-heap</td>
<td>5.04</td>
<td>4.67</td>
<td>4.84</td>
</tr>
<tr>
<td>5-heap</td>
<td>5.04</td>
<td>4.72</td>
<td>4.84</td>
</tr>
<tr>
<td>Binomial</td>
<td>5.04</td>
<td>5.56</td>
<td>4.84</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>5.04</td>
<td>5.84</td>
<td>4.84</td>
</tr>
<tr>
<td>Relaxed</td>
<td>5.04</td>
<td>6.54</td>
<td>4.84</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>5.04</td>
<td>3.95</td>
<td>4.84</td>
</tr>
<tr>
<td>2-level 512</td>
<td>5.04</td>
<td>3.97</td>
<td>4.84</td>
</tr>
<tr>
<td>3-level 64</td>
<td>5.04</td>
<td>4.04</td>
<td>4.84</td>
</tr>
<tr>
<td>4-level 24</td>
<td>5.04</td>
<td>4.10</td>
<td>4.84</td>
</tr>
<tr>
<td>5-level 12</td>
<td>5.04</td>
<td>4.20</td>
<td>4.84</td>
</tr>
<tr>
<td>6-level 8</td>
<td>5.04</td>
<td>4.25</td>
<td>4.84</td>
</tr>
<tr>
<td>radix</td>
<td>5.04</td>
<td>4.49</td>
<td>4.84</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>58.69</td>
<td>26.55</td>
<td>61.59</td>
</tr>
<tr>
<td>Parent check</td>
<td>37.57</td>
<td>20.36</td>
<td>38.43</td>
</tr>
<tr>
<td>LLL</td>
<td>34.83</td>
<td>13.23</td>
<td>33.60</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>25.82</td>
<td>12.05</td>
<td>25.14</td>
</tr>
<tr>
<td>d’Esopo-P.</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>70.24</td>
<td>21.29</td>
<td>67.70</td>
</tr>
<tr>
<td>LLL</td>
<td>59.35</td>
<td>15.71</td>
<td>54.49</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>24.15</td>
<td>11.43</td>
<td>23.89</td>
</tr>
<tr>
<td>LLL</td>
<td>20.20</td>
<td>8.34</td>
<td>19.06</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>17.82</td>
<td>10.14</td>
<td>17.57</td>
</tr>
<tr>
<td>Parent check</td>
<td>16.00</td>
<td>9.85</td>
<td>15.49</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>13.14</td>
<td>7.24</td>
<td>12.81</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>5.04</td>
<td>4.61</td>
<td>4.94</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>5.01</td>
<td>3.63</td>
<td>4.90</td>
</tr>
</tbody>
</table>
Figure 7.6: Unidirectional search (left) outperforms bidirectional search (right) on a hybrid preprocessed graph.

Table 7.29: Bidirectional methods (A*, symmetric) on the hybrid preprocessed graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th>Method</th>
<th>Dantzig</th>
<th>Pohl</th>
<th>Nicholson</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Iter.</td>
<td>CPU</td>
<td>Iter.</td>
</tr>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>4.74</td>
<td>4.47</td>
<td>4.53</td>
</tr>
<tr>
<td>3-heap</td>
<td>4.74</td>
<td>4.41</td>
<td>4.53</td>
</tr>
<tr>
<td>4-heap</td>
<td>4.74</td>
<td>4.39</td>
<td>4.53</td>
</tr>
<tr>
<td>5-heap</td>
<td>4.74</td>
<td>4.46</td>
<td>4.53</td>
</tr>
<tr>
<td>Binomial</td>
<td>4.74</td>
<td>5.21</td>
<td>4.53</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>4.74</td>
<td>5.39</td>
<td>4.53</td>
</tr>
<tr>
<td>Relaxed</td>
<td>4.74</td>
<td>6.16</td>
<td>4.53</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>4.74</td>
<td>3.63</td>
<td>4.54</td>
</tr>
<tr>
<td>2-level 512</td>
<td>4.74</td>
<td>3.71</td>
<td>4.53</td>
</tr>
<tr>
<td>3-level 64</td>
<td>4.74</td>
<td>3.77</td>
<td>4.53</td>
</tr>
<tr>
<td>4-level 24</td>
<td>4.74</td>
<td>3.82</td>
<td>4.53</td>
</tr>
<tr>
<td>5-level 12</td>
<td>4.74</td>
<td>3.92</td>
<td>4.53</td>
</tr>
<tr>
<td>6-level 8</td>
<td>4.74</td>
<td>3.95</td>
<td>4.53</td>
</tr>
<tr>
<td>radix</td>
<td>4.74</td>
<td>4.18</td>
<td>4.53</td>
</tr>
</tbody>
</table>
Table 7.30: Bidirectional methods (A*, scalar projections) on the hybrid preprocessed graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LS, heaps</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>4.72</td>
<td>4.44</td>
<td>4.62</td>
<td>4.06</td>
<td>4.79</td>
<td>4.47</td>
</tr>
<tr>
<td>3-heap</td>
<td>4.72</td>
<td>4.36</td>
<td>4.62</td>
<td>4.04</td>
<td>4.79</td>
<td>4.45</td>
</tr>
<tr>
<td>4-heap</td>
<td>4.72</td>
<td>4.40</td>
<td>4.62</td>
<td>4.04</td>
<td>4.79</td>
<td>4.45</td>
</tr>
<tr>
<td>5-heap</td>
<td>4.72</td>
<td>4.40</td>
<td>4.62</td>
<td>4.08</td>
<td>4.79</td>
<td>4.46</td>
</tr>
<tr>
<td>Binomial</td>
<td>4.72</td>
<td>5.17</td>
<td>4.62</td>
<td>4.82</td>
<td>4.79</td>
<td>5.61</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>4.72</td>
<td>5.36</td>
<td>4.62</td>
<td>4.91</td>
<td>4.79</td>
<td>5.40</td>
</tr>
<tr>
<td>Relaxed</td>
<td>4.72</td>
<td>6.15</td>
<td>4.62</td>
<td>5.69</td>
<td>4.79</td>
<td>6.98</td>
</tr>
<tr>
<td><strong>LS, buckets</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>4.72</td>
<td>3.63</td>
<td>4.63</td>
<td>3.39</td>
<td>4.79</td>
<td>3.73</td>
</tr>
<tr>
<td>2-level 512</td>
<td>4.72</td>
<td>3.69</td>
<td>4.62</td>
<td>3.45</td>
<td>4.79</td>
<td>3.79</td>
</tr>
<tr>
<td>3-level 64</td>
<td>4.72</td>
<td>3.79</td>
<td>4.62</td>
<td>3.53</td>
<td>4.79</td>
<td>3.86</td>
</tr>
<tr>
<td>4-level 24</td>
<td>4.72</td>
<td>3.82</td>
<td>4.62</td>
<td>3.57</td>
<td>4.79</td>
<td>3.90</td>
</tr>
<tr>
<td>5-level 12</td>
<td>4.72</td>
<td>3.90</td>
<td>4.62</td>
<td>3.62</td>
<td>4.79</td>
<td>3.98</td>
</tr>
<tr>
<td>6-level 8</td>
<td>4.72</td>
<td>3.92</td>
<td>4.62</td>
<td>3.68</td>
<td>4.79</td>
<td>4.05</td>
</tr>
<tr>
<td>radix</td>
<td>4.72</td>
<td>4.16</td>
<td>4.62</td>
<td>3.91</td>
<td>4.79</td>
<td>4.28</td>
</tr>
<tr>
<td><strong>BFM</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>138.86</td>
<td>87.45</td>
<td>137.05</td>
<td>74.03</td>
<td>142.37</td>
<td>77.53</td>
</tr>
<tr>
<td>Parent check</td>
<td>82.05</td>
<td>59.39</td>
<td>84.49</td>
<td>53.18</td>
<td>84.64</td>
<td>52.70</td>
</tr>
<tr>
<td>LLL</td>
<td>39.81</td>
<td>16.37</td>
<td>37.89</td>
<td>14.09</td>
<td>35.68</td>
<td>13.28</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>28.33</td>
<td>14.20</td>
<td>29.32</td>
<td>13.63</td>
<td>26.36</td>
<td>12.33</td>
</tr>
<tr>
<td><strong>d’Esopo-P.</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>121.18</td>
<td>42.22</td>
<td>121.94</td>
<td>39.98</td>
<td>121.86</td>
<td>40.24</td>
</tr>
<tr>
<td>LLL</td>
<td>72.88</td>
<td>20.69</td>
<td>67.68</td>
<td>17.79</td>
<td>65.37</td>
<td>18.12</td>
</tr>
<tr>
<td><strong>Pallottino</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>38.79</td>
<td>19.98</td>
<td>38.73</td>
<td>18.13</td>
<td>40.34</td>
<td>19.05</td>
</tr>
<tr>
<td>LLL</td>
<td>24.14</td>
<td>10.77</td>
<td>23.98</td>
<td>9.91</td>
<td>22.57</td>
<td>9.70</td>
</tr>
<tr>
<td><strong>SLP</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>28.28</td>
<td>18.84</td>
<td>28.72</td>
<td>17.38</td>
<td>29.23</td>
<td>17.63</td>
</tr>
<tr>
<td>Parent check</td>
<td>25.96</td>
<td>19.21</td>
<td>26.04</td>
<td>17.18</td>
<td>26.46</td>
<td>17.37</td>
</tr>
<tr>
<td>LLL</td>
<td>15.78</td>
<td>8.19</td>
<td>15.75</td>
<td>7.65</td>
<td>15.57</td>
<td>7.73</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>15.19</td>
<td>8.69</td>
<td>14.73</td>
<td>7.79</td>
<td>14.26</td>
<td>7.53</td>
</tr>
<tr>
<td><strong>Other</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>4.79</td>
<td>4.12</td>
<td>4.75</td>
<td>3.91</td>
<td>4.82</td>
<td>4.04</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>4.72</td>
<td>3.39</td>
<td>4.60</td>
<td>3.17</td>
<td>4.79</td>
<td>3.44</td>
</tr>
</tbody>
</table>
Table 7.31: Bidirectional methods (Landmarks, balanced) on the hybrid preprocessed graph. Cumulated running times (seconds) and iterations (in millions).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>1.13</td>
<td>1.36</td>
<td>1.15</td>
<td>1.25</td>
<td>1.12</td>
<td>1.23</td>
</tr>
<tr>
<td>3-heap</td>
<td>1.13</td>
<td>1.32</td>
<td>1.15</td>
<td>1.24</td>
<td>1.12</td>
<td>1.23</td>
</tr>
<tr>
<td>4-heap</td>
<td>1.13</td>
<td>1.33</td>
<td>1.15</td>
<td>1.23</td>
<td>1.12</td>
<td>1.21</td>
</tr>
<tr>
<td>5-heap</td>
<td>1.13</td>
<td>1.33</td>
<td>1.15</td>
<td>1.24</td>
<td>1.12</td>
<td>1.22</td>
</tr>
<tr>
<td>Binomial</td>
<td>1.13</td>
<td>1.47</td>
<td>1.16</td>
<td>1.38</td>
<td>1.12</td>
<td>1.41</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>1.13</td>
<td>1.55</td>
<td>1.15</td>
<td>1.43</td>
<td>1.12</td>
<td>1.41</td>
</tr>
<tr>
<td>Relaxed</td>
<td>1.13</td>
<td>1.68</td>
<td>1.17</td>
<td>1.60</td>
<td>1.12</td>
<td>1.71</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>1.13</td>
<td>1.19</td>
<td>1.17</td>
<td>1.13</td>
<td>1.12</td>
<td>1.08</td>
</tr>
<tr>
<td>2-level 512</td>
<td>1.13</td>
<td>1.22</td>
<td>1.17</td>
<td>1.12</td>
<td>1.12</td>
<td>1.09</td>
</tr>
<tr>
<td>3-level 64</td>
<td>1.13</td>
<td>1.23</td>
<td>1.17</td>
<td>1.12</td>
<td>1.12</td>
<td>1.09</td>
</tr>
<tr>
<td>4-level 24</td>
<td>1.13</td>
<td>1.21</td>
<td>1.17</td>
<td>1.14</td>
<td>1.12</td>
<td>1.11</td>
</tr>
<tr>
<td>5-level 12</td>
<td>1.13</td>
<td>1.21</td>
<td>1.17</td>
<td>1.15</td>
<td>1.12</td>
<td>1.12</td>
</tr>
<tr>
<td>6-level 8</td>
<td>1.13</td>
<td>1.22</td>
<td>1.17</td>
<td>1.16</td>
<td>1.12</td>
<td>1.12</td>
</tr>
<tr>
<td>radix</td>
<td>1.13</td>
<td>1.30</td>
<td>1.17</td>
<td>1.20</td>
<td>1.12</td>
<td>1.16</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>20.60</td>
<td>13.76</td>
<td>19.70</td>
<td>11.58</td>
<td>23.81</td>
<td>14.26</td>
</tr>
<tr>
<td>Parent check</td>
<td>12.73</td>
<td>10.28</td>
<td>11.59</td>
<td>8.58</td>
<td>12.91</td>
<td>9.77</td>
</tr>
<tr>
<td>LLL</td>
<td>8.46</td>
<td>5.22</td>
<td>7.31</td>
<td>4.56</td>
<td>7.96</td>
<td>4.96</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>6.48</td>
<td>5.00</td>
<td>5.38</td>
<td>4.09</td>
<td>5.83</td>
<td>4.53</td>
</tr>
<tr>
<td>d'Esopo-P.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>11.69</td>
<td>8.32</td>
<td>9.46</td>
<td>6.95</td>
<td>12.71</td>
<td>8.52</td>
</tr>
<tr>
<td>LLL</td>
<td>7.45</td>
<td>4.50</td>
<td>6.14</td>
<td>3.91</td>
<td>6.62</td>
<td>4.17</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>6.29</td>
<td>6.57</td>
<td>6.06</td>
<td>5.73</td>
<td>6.80</td>
<td>6.47</td>
</tr>
<tr>
<td>LLL</td>
<td>4.29</td>
<td>3.83</td>
<td>3.93</td>
<td>3.40</td>
<td>4.08</td>
<td>3.57</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>3.03</td>
<td>2.40</td>
<td>3.16</td>
<td>2.29</td>
<td>3.34</td>
<td>2.46</td>
</tr>
<tr>
<td>Parent check</td>
<td>2.53</td>
<td>2.29</td>
<td>3.08</td>
<td>2.35</td>
<td>3.19</td>
<td>2.48</td>
</tr>
<tr>
<td>LLL</td>
<td>2.33</td>
<td>1.94</td>
<td>2.64</td>
<td>2.00</td>
<td>2.79</td>
<td>2.18</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>2.21</td>
<td>1.94</td>
<td>2.49</td>
<td>1.99</td>
<td>2.59</td>
<td>2.15</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>1.30</td>
<td>1.55</td>
<td>1.28</td>
<td>1.44</td>
<td>1.29</td>
<td>1.49</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>1.16</td>
<td>1.07</td>
<td>1.33</td>
<td>1.21</td>
<td>1.24</td>
<td>1.15</td>
</tr>
</tbody>
</table>
7.6 Contraction Hierarchies

We implemented the Contraction Hierarchies preprocessing. As described in Section 6.4.1 a node ordering ‘<’ is needed. We used the node ordering that is proposed by Abraham et al. [1]. The priority key of each node $v \in V$ is set to: $2AD(v) + CN(v) + H(v) + 5L(v)$. Here, $AD(v)$ is the difference between the number of arcs added and removed if node $v$ is bypassed. The number of neighbors of node $v$ that are already bypassed is stored as $CN(v)$. The number of original arcs that are represented by the shortcuts that have to be added to bypass node $v$, is stored as $H(v)$ (the so-called ‘hop’ number). Finally, $L(v)$ is initialized to zero for all nodes $v \in V$. Each time a node $u$ is bypassed, for all neighbors $v$ of node $u$ the value $L(v)$ is updated to $\max\{L(v), L(u) + 1\}$. Each time a node $u$ is bypassed the priority keys of its neighbors have to be updated.

To solve the SPP on a graph that results after Contraction Hierarchies are applied, a bidirectional search is executed, where the so-called upward graph $G^U = (V, A^U)$ is used in the forward search and the downward graph $G^D = (V, A^D)$ in the backward search (see Section 6.4.2). When both searches meet each other at node $u$ the process cannot be stopped, but the tentative path length $d^u_s + d^u_t$ can be used to prune the search in each direction. If node $v$ is selected by the forward search and $d^u_s \geq d^u_s + d^u_t$ the forward search can be stopped in a label-setting environment. In a label-correcting environment, node $v$ can be skipped, but the forward search itself has to be continued. A similar argument holds for the backward search. Hence, it is possible to use both label-setting and label-correcting algorithms on the graph that results after executing the Contraction Hierarchies preprocessing step.

Table 7.32 shows the results for each optimization algorithm and for the three different alternation rules for forward and backward search. Note that the numbers in these tables are in milliseconds (CPU) and in thousands (iterations). Clearly all algorithms can be considered as fast. However, the following observations can be made. First, the alternation rule of Nicholson reduces the number of iterations for each label-setting algorithm and even for some label-correcting algorithms. As stated before, the alternation rule of Nicholson where the search is continued in the direction that has a top-node in $Q$ with the smallest label (compared to the other direction) can only be emulated by label-correcting algorithms. In all other tables Nicholson’s rule was outperformed by the other ones. The effectiveness of Nicholson’s rule here, can easily be explained. Since the search can only be pruned after a meeting node is found, one should try to find a meeting node as early as possible. This can be done by exploring both search spaces as much as possible at the same hierarchy level. In the forward search we have $d^v_s \leq d^w_s$ for each $v < w$, and similarly, in the backward search $d^v_t \leq d^w_t$ for each $v < w$. Therefore, Nicholson’s alternation rule naturally tries to reduce the hierarchy level gap between the forward search and backward search. However, since the number of iterations is already low for both other alternation rules, the effort to read and compare the node keys of each direction outweighs the reduction of the number of iterations.
In order to decrease the running times even further, one can try to optimize the storage of the graph and the order in which operations are executed such that the number of so-called ‘cache-misses’ found by the machine that executes the algorithm is minimized [1, 48, 51]. We did not implement such optimizations here.

### Table 7.32: Contraction Hierarchies. Cumulated running times (milliseconds) and iterations (in thousands).

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>LS, heaps</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Binary heap</td>
<td>112.0</td>
<td>70.4</td>
<td>113.2</td>
<td>65.6</td>
<td>111.9</td>
<td>67.5</td>
</tr>
<tr>
<td>3-heap</td>
<td>112.0</td>
<td>68.4</td>
<td>113.2</td>
<td>65.2</td>
<td>111.9</td>
<td>67.7</td>
</tr>
<tr>
<td>4-heap</td>
<td>112.0</td>
<td>66.6</td>
<td>113.2</td>
<td>65.6</td>
<td>111.9</td>
<td>69.6</td>
</tr>
<tr>
<td>5-heap</td>
<td>112.0</td>
<td>67.4</td>
<td>113.2</td>
<td>65.5</td>
<td>111.9</td>
<td>68.5</td>
</tr>
<tr>
<td>Binomial</td>
<td>112.0</td>
<td>76.9</td>
<td>113.2</td>
<td>75.6</td>
<td>111.9</td>
<td>83.9</td>
</tr>
<tr>
<td>Fibonacci</td>
<td>112.0</td>
<td>84.6</td>
<td>113.2</td>
<td>83.9</td>
<td>111.9</td>
<td>87.4</td>
</tr>
<tr>
<td>Relaxed</td>
<td>112.0</td>
<td>95.3</td>
<td>113.2</td>
<td>96.8</td>
<td>111.9</td>
<td>110.9</td>
</tr>
<tr>
<td>LS, buckets</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Dial</td>
<td>112.0</td>
<td>137.6</td>
<td>113.2</td>
<td>136.8</td>
<td>111.9</td>
<td>143.5</td>
</tr>
<tr>
<td>2-level 512</td>
<td>112.0</td>
<td>79.6</td>
<td>113.2</td>
<td>78.8</td>
<td>111.9</td>
<td>80.1</td>
</tr>
<tr>
<td>3-level 64</td>
<td>112.0</td>
<td>65.9</td>
<td>113.2</td>
<td>66.0</td>
<td>111.9</td>
<td>68.4</td>
</tr>
<tr>
<td>4-level 24</td>
<td>112.0</td>
<td>66.8</td>
<td>113.2</td>
<td>67.8</td>
<td>111.9</td>
<td>69.6</td>
</tr>
<tr>
<td>5-level 12</td>
<td>112.0</td>
<td>75.8</td>
<td>113.2</td>
<td>70.8</td>
<td>111.9</td>
<td>72.4</td>
</tr>
<tr>
<td>6-level 8</td>
<td>112.0</td>
<td>72.0</td>
<td>113.2</td>
<td>71.8</td>
<td>111.9</td>
<td>87.7</td>
</tr>
<tr>
<td>radix</td>
<td>112.0</td>
<td>72.7</td>
<td>113.2</td>
<td>72.8</td>
<td>111.9</td>
<td>73.2</td>
</tr>
<tr>
<td>BFM</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>185.8</td>
<td>76.0</td>
<td>188.1</td>
<td>75.9</td>
<td>185.9</td>
<td>76.2</td>
</tr>
<tr>
<td>Parent check</td>
<td>142.9</td>
<td>70.7</td>
<td>144.3</td>
<td>69.4</td>
<td>143.0</td>
<td>70.3</td>
</tr>
<tr>
<td>LLL</td>
<td>128.5</td>
<td>62.2</td>
<td>130.0</td>
<td>63.4</td>
<td>128.5</td>
<td>63.0</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>120.2</td>
<td>64.5</td>
<td>121.4</td>
<td>64.0</td>
<td>120.2</td>
<td>64.7</td>
</tr>
<tr>
<td>d’Esopo-P.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>127.5</td>
<td>60.1</td>
<td>128.9</td>
<td>58.0</td>
<td>127.4</td>
<td>59.8</td>
</tr>
<tr>
<td>LLL</td>
<td>118.4</td>
<td>59.3</td>
<td>119.4</td>
<td>58.8</td>
<td>118.4</td>
<td>58.3</td>
</tr>
<tr>
<td>Pallottino</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>127.0</td>
<td>59.4</td>
<td>128.4</td>
<td>57.8</td>
<td>127.0</td>
<td>58.9</td>
</tr>
<tr>
<td>LLL</td>
<td>118.3</td>
<td>58.8</td>
<td>119.4</td>
<td>59.9</td>
<td>118.3</td>
<td>60.0</td>
</tr>
<tr>
<td>SLF</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Basic</td>
<td>163.2</td>
<td>71.7</td>
<td>164.9</td>
<td>70.6</td>
<td>163.2</td>
<td>71.9</td>
</tr>
<tr>
<td>Parent check</td>
<td>136.2</td>
<td>69.8</td>
<td>137.7</td>
<td>68.0</td>
<td>136.2</td>
<td>70.0</td>
</tr>
<tr>
<td>LLL</td>
<td>124.1</td>
<td>63.9</td>
<td>125.3</td>
<td>61.6</td>
<td>124.1</td>
<td>62.7</td>
</tr>
<tr>
<td>LLL and PC</td>
<td>118.5</td>
<td>65.9</td>
<td>119.6</td>
<td>62.7</td>
<td>118.4</td>
<td>65.3</td>
</tr>
<tr>
<td>Other</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Threshold</td>
<td>112.0</td>
<td>128.5</td>
<td>113.2</td>
<td>128.5</td>
<td>112.0</td>
<td>131.1</td>
</tr>
<tr>
<td>Approx. b.</td>
<td>112.1</td>
<td>54.9</td>
<td>113.6</td>
<td>54.0</td>
<td>112.0</td>
<td>55.6</td>
</tr>
</tbody>
</table>

The Hub Based Labeling algorithm of Abraham et al. [1] leads to even faster running times at the expense of memory storage. Since our test environment was limited to 16Gb of memory, we were not able to implement this algorithm. As a comparison, we mention that in [1] an average time for solving an SSSD-SPP is reported of less than 1 µs for a graph that is approximately twice as big as our test graph.
7.7 Concluding remarks

In this chapter we presented the results of 28 optimization methods that were ran in a variety of different ways. Based on these experiments we now draw some conclusions.

- If Dijkstra is implemented with a heap, one should always consider the ordinary binary heap. Although there are more sophisticated heaps, like the Fibonacci-heap and the Relaxed heap which have theoretically better worst-case bounds, the binary heap turns out to be faster in all our experiments.

- If arc costs are integral one should consider to implement Dijkstra with Dial’s bucket data structure.

- A suitable label-correcting algorithm that turns out to be the fastest in all our experiments, except for the ones that used a landmark estimator, is the Approximate buckets method. It is closely related to Dijkstra implemented with Dial’s method, but can also be applied for real-valued arc costs.

- Applying the Euclidean estimator in a unidirectional way improves the efficiency with about 60%.

- If a landmark-based estimator is applied to a unidirectional method, one should consider to run a Depth First Search from the destination node, as described in Section 7.2.1.1.

- Bidirectional search turns out be more efficient for all our experiments, except for the ones that were conducted on a hybrid graph.

- Pohl’s alternation rule for choosing the active search direction (forward or backward) can be considered to be the most effective. Only bidirectional methods that use a landmark estimator profit more from Nicholson’s alternation rule. Although the number of iterations on the graph that was preprocessed with the Contraction Hierarchies procedure is the smallest when Nicholson’s rule is applied, applying Pohl’s alternation rule still leads to faster running times in our experiments.

- If a Euclidean estimator is used in a bidirectional way, the symmetric approach that uses our new bound leads to slightly faster implementations than the balanced approach.

- The fastest bidirectional methods that use a Euclidean estimator are those variants that apply the estimator in a symmetric way combined with scalar projections to divide the search in the post-phase of the algorithm.

- The landmark-based estimator is not suitable for use in a bidirectional, symmetric approach.

- The landmark-based estimator that is used in a bidirectional balanced approach leads to the best results for all our graphs, except for the one that results after hybrid preprocessing.
Applying the static preprocessing as described above, results in a graph with a set of core nodes of about 31% of the number of original nodes. Speedups of about the same fraction are found in our experiments.

Applying the dynamic preprocessing as described above, leads to the expected decrease in running time of about 97% of most methods tested.

For a hybrid preprocessed graph, where the number of core-nodes is much smaller than the number of nodes in each cell of the graph partition, unidirectional methods outperformed their bidirectional counterparts, no matter which type of bidirectional search is used.

Among all our experiments, the results on the graph that was preprocessed with Contraction Hierarchies were the best. However, we already mentioned that each preprocessing we mentioned has its own characteristics that makes it suitable for specific situations. Sometimes preprocessing will not lead to any improvement, as stated in Section 7.2.3. Static preprocessing not only is very fast, but updates in arc costs can be done very efficiently, since only the arc costs of some shortcuts should be changed. If the arc costs of $m$ arcs have to be updated this can be done in $O(m)$ time, by maintaining a pointer for each original arc towards NIL or the unique shortcut that contains this arc. Dynamic preprocessing results in fast results, but is less suited for updating arc costs. The same holds for Contraction Hierarchies. Dynamic preprocessing and Contraction Hierarchies might store the order in which nodes are bypassed, to be reused as soon as new arc costs are provided. This might, however, not only lead to a defect in the quality of the preprocessing itself, but the actual bypassing has to be executed for each update.

Hybrid preprocessing combines the best of two worlds. The number of nodes that has to be considered is even smaller than in our dynamically preprocessed graph. Furthermore, as soon as some arc costs are updated, only the shortcuts between each pair of border nodes of a cell should be reetermined. This can be done within a minute, especially if we first apply the static preprocessing to the graph, such that the shortest path tree that has to be grown from a border node of a cell does not consider ‘quarters’ or ‘degree 2’ nodes. Note that both type of nodes are surely not border nodes of the cell.

Clearly, the landmark estimator outperforms the Euclidean estimator in all our experiments. However, there might be several reasons to use the Euclidean estimator. Since it only uses the coordinates of each node which will not change during the ‘life-cycle’ of the graph, there is no need for updating something as soon as new arc costs are applied. Furthermore, since in several applications these coordinates are already included in the data, no extra memory is needed. The distances from and to a landmark have to be stored and to be updated as soon as new arc costs are applied.
Abbreviations

**ADT** Abstract Data Type. A model of a class of data structures. This model consists only of the supported operations by the data structures in that class.

**APSP** All Pairs Shortest Path Problem. In this problem, the shortest path between every pair of nodes has to be determined.

**CS conditions** Complementary slackness conditions.

**CSP** Constraint Satisfaction Problem. A mathematical problem defined as a set of variables, whose values must satisfy a number of constraints.

**DPC** Directional Path-Consistency.

**MCFP** Minimum Cost Flow Problem.

**MSCP** Minimum Set Cover Problem.

**MSMD-SPP** Multiple-Source, Multiple-Destination Shortest Path Problem. A special variant of the *SPP* to determine a time and/or distance matrix between a set of origin nodes and a set of destination nodes.

**NCDP** Negative Cycle Detection Problem. The problem of verifying whether or not a graph contains a cycle of negative length.

**SPP** The Shortest Path Problem. In its most basic form referring to the problem of finding the shortest path from a single origin to all other nodes.

**SSMD-SPP** Single-Source, Multiple-Destination Shortest Path Problem. This variant of the *SPP* is most commonly used in cases where the number of destinations is considerably smaller than the number of nodes in the graph.

**SSSD-SPP** Single-Source, Single-Destination Shortest Path Problem. A special variant of the *SPP* in which the shortest path from one node to a certain other node has to be determined.
**STP** Simple Temporal Problem. A special case of the *TSCP*.

**TDSPP** Time-Dependent Shortest Path Problem. A generalization of the *SPP* where the arc costs are considered to be time-dependent.

**TSCP** Temporal Constraint Satisfaction Problem. A special case of the *CSP*. 
In label-setting algorithms (see Section 3.2), the candidate set \( Q \) is maintained as a priority queue (see Definition 3.8). Algorithm B.1 describes Dijkstra’s algorithm with a priority queue.

**Algorithm B.1: Dijkstra using a priority queue \((G, s, t)\)**

1. \( Q . \text{Create} \)
2. foreach \( v \in V \) do
3. \( d_v \leftarrow \infty \)
4. \( \pi_v \leftarrow \text{NIL} \)
5. \( d_s \leftarrow 0 \)
6. \( Q . \text{Insert}(v, d_v) \)
7. repeat
8. \( Q . \text{Delete-Min}(v) \)
9. if \( v \neq t \) then
10. foreach \( (v, w) \in \delta^+(v) \) do
11. if \( d_v + c_{vw} < d_w \) then
12. if \( w \notin Q \) then
13. \( Q . \text{Insert}(w, d_v + c_{vw}) \)
14. else
15. \( Q . \text{Decrease-Key}(w, d_v + c_{vw}) \)
16. \( d_w \leftarrow d_v + c_{vw} \)
17. \( \pi_w \leftarrow v \)
18. until \( v = t \)

A priority queue is defined by the following operations that can be performed on it:

- **CREATE.** Return a new, empty priority queue.
- **INSERT\((v, k)\).** Add an item \( v \) with a predefined key \( k \) to the priority queue.
• \textbf{FIND-MIN}. Return an item $v$ with the smallest key among all items in the priority queue.

• \textbf{DELETE-MIN}(v). Delete an item $v$ with the smallest key among all items in the priority queue and return $v$.

• \textbf{DECREASE-KEY}(v,k). Assign the new (lower) key $k$ to item $v$.

From Section 3.2.2, we know that during the execution of Algorithm B.1, each node $v \in \mathcal{V}$ is added to $\mathcal{Q}$ at most once, and each node $v \in \mathcal{V}$ is selected as a pivot node at most once. Thus the operations \textsc{Insert} and \textsc{Delete-Min} are performed at most $|\mathcal{V}|$ times. The operation \textsc{Decrease-Key} is performed at most $|\mathcal{A}|$ times. Clearly, the operation \textsc{Create} is performed once. Although Algorithm B.1 does not use the operation \textsc{Find-Min}, this operation is needed. The alternation rule of Nicholson (see Section 4.2.2) for example, needs to find the minimum label among all nodes in $\mathcal{Q}$ without deleting a node $v$ with such a minimum label.

As stated in Proposition 3.6, the running time of Dijkstra’s algorithm is bounded by $O(|\mathcal{V}|^2 + |\mathcal{A}|)$. The worst-case bound on the running time of Dijkstra’s algorithm can be reduced if $|\mathcal{A}| = O(|\mathcal{V}|)$, by using a priority queue for implementing $\mathcal{Q}$.

In this appendix, we will describe some data structures that implement a priority queue.

\section*{B.1 Heaps}

A heap is a well-known data structure to implement a priority queue. We will discuss several different heaps in this section. Most heaps maintain the set $\mathcal{Q}$ as either an ordered tree, or as a forest of ordered trees.

\textbf{Definition B.1} Let $\mathcal{T}$ be a rooted tree. The depth of a node $v \in \mathcal{T}$ is defined as the number of arcs in the unique path from the source of $\mathcal{T}$ to node $v$.

By taking care of the maximum depth of each tree in the heap, such that it never exceeds $O(\log |\mathcal{V}|)$, the priority queue operations are bounded by $O(\log |\mathcal{V}|)$ as well, as we will see later. Knuth [124] stated that many new ways to represent priority queues have been discovered nowadays and that not all of them will survive the test of time. As an example of already obsolete heaps, he mentioned the so-called \textit{leftist heap}, discovered by Crane [36] and modified by Knuth himself. In this section, we will restrict ourselves to only discuss some of the most commonly used heaps.

\subsection*{B.1.1 d-Heap}

The heap data structure was first introduced by Williams [199] and improved by Floyd [69]. Their heap is nowadays referred to as \textit{binary heap}. They used the heap in a sorting algorithm, currently known as \textit{heap sort}. Johnson [114] introduced the
so-called \( d \)-heap, where \( d \in \mathbb{N} \) and \( d \geq 2 \). The binary heap is a special case of the \( d \)-heap (where \( d = 2 \)).

In a \( d \)-heap implementation of \( Q \), the nodes of \( Q \) are maintained as a rooted tree \( T \) (see Definition 2.47). The arcs of this tree represent a predecessor-successor relationship. Each node in \( T \) has at most \( d \) successors. Figure B.1 [3] gives an example of a 3-heap. In this heap, node 5 has depth 0, and node 20 has depth 3.

![Figure B.1: Example of a \( d \)-heap for \( d = 3 \)](image)

There is no need to store pointers in memory: Neither to relate a particular node to its predecessor nor to relate a particular node to its successors [125]. Based on the so-called contiguity property of the \( d \)-heap, the predecessor of a node and the successors of a node can easily be determined by some simple calculation.

**Definition B.2** A \( d \)-heap always satisfies the contiguity property, which means that nodes are stored in increasing order of depth values and for the same depth value, nodes are added from left to right.

**Proposition B.3** [3] The contiguity property of a \( d \)-heap implies:

\begin{enumerate}
\item At most \( d^k \) nodes have depth \( k \).
\item At most \( (d^{k+1} - 1)/(d - 1) \) nodes have a depth of between 0 and \( k \).
\item The depth of a \( d \)-heap containing \( n \) nodes, is at most \( \lceil \log_d n \rceil \).\(^1\)
\end{enumerate}

**Proof**: We prove (i) by induction. Clearly, (i) holds for depth 0, since at most \( d^0 = 1 \) nodes can be stored at this depth. Now assume that (i) holds for a certain depth \( k \geq 0 \). Thus, we assume there are at most \( d^k \) nodes at depth \( k \). Each of these nodes can have at most \( d \) children. At depth \( k + 1 \), at most \( d^{k+1} \) nodes are stored. This completes the induction proof. From (i), it follows that at most \( d^0 + d^1 + \ldots + d^k = (d^{k+1} - 1)/(d - 1) \) nodes have a depth of between 0 and \( k \). This

\(^1\)Originally it was stated that the depth is at most \( \lfloor \log_d n \rfloor + 1 \). We slightly improved it.
is exactly what is stated in (ii). Let \( k \) be the maximum depth of the \( d \)-heap, which contains \( n \) nodes. From (i) and (ii), we have \( d^0 + d^1 + \ldots + d^{k-1} < n \leq d^0 + d^1 + \ldots + d^k \). Thus, \( \log_d n > \log_d(d^0 + d^1 + \ldots + d^{k-1}) > \log_d d^{k-1} = k - 1 \). Hence, \( \lceil \log_d n \rceil \geq k \), which completes the proof.

Johnson [114] stated that a \( d \)-heap can be stored as an array, if the maximum number of elements in the heap is known in advance. In Algorithm B.1, the number of nodes in the \( d \)-heap is at most \( |V| \). For each node \( v \in Q \), we store the position \( v_p \geq 0 \) in the heap-array of that node. If node \( v \notin Q \), then \( v_p \leftarrow -1 \). We use the contiguity property of Definition B.2 to determine the position in the heap-array of a node. For example, the heap of Figure B.1 is stored as [5, 9, 8, 15, 21, 12, 16, 18, 29, 10, 31, 22, 27, 28, 36, 32, 14, 13, 20, 38, 17, 41, 52, 42, 48, 39].

**Proposition B.4** [3] The contiguity property of a \( d \)-heap containing \( n \) nodes implies:

(i) The successors of the node in position \( p \) are contained in positions \( pd - d + 1, pd - d + 2, \ldots, \min(n, pd) \).

(ii) The predecessor of the node in position \( p \) is contained in position \( \lceil p/d \rceil \).

**Proof:** We prove (i) by induction. The children of the node on position 0 are stored in positions 1, 2, \ldots, \( \min(n, d) \), so (i) holds for \( p = 0 \). Assume that (i) holds for any \( p \geq 0 \). The children of the node on position \( p + 1 \) are stored directly (due to the contiguity property) after the children of the node on position \( p \), which by assumption are stored in positions \( pd - d + 1, pd - d + 2, \ldots, \min(n, pd) \). Thus, the children of the node on position \( p + 1 \) are stored on positions \( pd + 1, \ldots, pd + 2, \ldots, \min(n, pd + d) \). These positions can be rewritten to \( (p + 1)d - d + 1, (p + 1)d - d + 2, \ldots, \min(n, (p + 1)d) \). This completes the induction proof.

Based on (i), the children of the node in position \( x \) are stored in positions \( xd - d + 1, xd - d + 2, \ldots, \min(n, xd) \). Choose \( x \) such that \( p \in [xd - d + 1, \min(n, xd)] \). The parent of any node in array position \( y \) with \( y \in [xd - d + 1, \min(n, xd)] \) is placed in position \( x \). For any value \( y \in [xd - d + 1, \min(n, xd)] \) we have \( \lceil y/d \rceil = x \). Thus, the parent of the node in position \( p \) is stored in position \( \lceil p/d \rceil \).

**Definition B.5** A \( d \)-heap maintains the following heap order invariant. The label \( d_v \) of node \( v \) in the heap is less than, or equal to, the label of each of its successors in the heap.

In Figure B.1, the heap order invariant holds if we take \( d_v = v \). During the execution of a specific heap operation, this heap order invariant might be violated but at the end of the operation, the invariant is re-established. Before we describe the heap operations in more detail, we discuss four subroutines for a \( d \)-heap.

- **Swap** (\( v, w \))
  Nodes \( v \) and \( w \) are swapped in the \( d \)-heap. This can easily be done in \( O(1) \) time...
by swapping the values in the array that represents the heap in the positions \( v_p \) and \( w_p \), followed by swapping the values \( v_p \) and \( w_p \) itself.

- **Siftup\((v)\)**
  Assume that \( w \) is the predecessor of node \( v \) in the heap and that \( d_v < d_w \). To restore the heap order invariant, nodes \( v \) and \( w \) are swapped. This ‘sift-up’ of node \( v \) is continued until either node \( v \) is the source of \( T \), or \( d_v \geq d_w \), where \( w \) is the predecessor of node \( v \) in \( T \). Since the maximum depth in the \( d \)-heap is \( \lceil \log d \cdot n \rceil \) (see Proposition B.3), the Siftup subroutine takes \( O(\log_d n) \) time.

- **MinChild\((v)\)**
  Return node \( w \) with the minimum label among all (direct) successors of node \( v \) in the \( d \)-heap. Clearly, this subroutine runs in \( O(d) \) time.

- **Siftdown\((v)\)**
  Let \( w \leftarrow \text{MinChild}(v) \). If \( d_v > d_w \), the heap order invariant has to be restored by swapping \( v \) and \( w \). The ‘sift-down’ of node \( v \) is continued until either node \( v \) is a sink (leaf) of \( T \) or \( d_v \leq d_w \), for any successor \( w \) of node \( v \). The execution of this subroutine takes \( O(d \cdot \log_d n) \) time.

The **Insert\((v, k)\)** operation is implemented as follows. First, the number of elements \( n \) in the \( d \)-heap is increased by one. Next, \( v_p \leftarrow n \) and \( v \) is stored in position \( n \) of the heap-array. To restore the (possibly violated) heap order invariant the subroutine Siftup\((v)\) is executed. Thus, the operation Insert takes \( O(\log_d |V|) \) time.

The **Delete-Min\((v)\)** operation consists of the following steps. First, node \( v \) is the node in position 0 of the heap-array. Let \( w \) be the node that is in position \( n \) of the array. The subroutine Swap\((v, w)\) is executed, after which the number of nodes \( n \) in the heap is decreased by one. The subroutine Siftdown\((w)\) is executed to restore the heap order invariant. The operation Delete-Min takes \( O(d \cdot \log_d |V|) \) time. During the execution of the **Decrease-Key\((v, k)\)** operation the node label is updated: \( d_v \leftarrow k \). To restore the heap order invariant, the Siftup\((v)\) subroutine is executed. The operation Decrease-Key takes therefore \( O(\log_d |V|) \) time.

**Proposition B.6** The running time of Algorithm B.1 using a \( d \)-heap data structure is \( O(d|V| \cdot \log_d |V| + |A| \cdot \log_d |V|) \).

**Proof:** The operation Create takes \( O(1) \) time (the amount of elements \( n \) in the heap is set to zero). The operations Insert and Decrease-Key take \( O(\log_d |V|) \) time as stated above. The operation Delete-Min takes \( O(d \cdot \log_d |V|) \) time. Since the operations Insert and Delete-Min are executed \( O(|V|) \) times and the operation Decrease-Key is executed \( O(|A|) \) times, the total running time is \( O(d|V| \cdot \log_d |V| + |A| \cdot \log_d |V|) \). □

The running time as stated in Proposition B.6 can be rewritten as \( O(f(n, m, d) + g(n, m, d)) \), where \( n \) represents the number of nodes, \( m \) the number of arcs, and \( d \in \mathbb{N} \).
206 B Priority queue

with \(d \geq 2\). The function \(f(n, m, d) = dn \log_d n\) is monotonically increasing in \(d\) and the function \(g(n, m, d) = m \log_d n\) is monotonically decreasing in \(d\). Furthermore \(f(n, m, d) \geq 0\) and \(g(n, m, d) \geq 0\), since both \(n\) and \(m\) are positive. Thus, we can use parameter balancing to determine a value \(d^*\) for \(d\), such that the resulting complexity is no bigger than twice the optimal complexity for any \(d\)-heap implementation. From Proposition 2.18, we search for \(d^*\) such that \(f(n, m, d^*) = g(n, m, d^*)\):

\[
d^*n \log_{d^*} n = m \log_{d^*} n.
\]

Thus, \(d^* = m/n\). Since we need \(d \in \mathbb{N}\) and \(d \geq 2\), we take \(d = \max\{2, [\lfloor A \rfloor / |V|]\}\).

### B.1.2 Binomial heaps

Binomial heaps are proposed by Vuillemin [194] and analyzed in detail by Brown [24]. Binomial heaps are motivated by the need of an extra operation on priority queues, at top of the five already mentioned at the beginning of this appendix. This additional operation is \(\text{UNION}(\mathcal{H}', \mathcal{H}'')\), which merges two priority queues \(\mathcal{H}'\) and \(\mathcal{H}''\). We will discuss the binomial heap in more detail, since the binomial heap provides the basis for other more sophisticated heaps.

**Definition B.7** [35] An ordered tree is a rooted tree in which the children of each node are ordered. That is, if a node has \(d\) children, then there is a first child, a second child, \ldots, and a \(d\text{th} \) child.

**Definition B.8** A binomial tree \(B_k\) is an ordered tree defined recursively (see Figure B.2(a)). The binomial tree \(B_0\) consists of a single node. The binomial tree \(B_k\) consists of two binomial trees \(B_{k-1}\), linked together. The root of one is the leftmost child of the root of the other.

In Figure B.2(b) [35, 194], the binomial trees \(B_0, B_1, B_2, B_3\) and \(B_4\) are shown.

**Proposition B.9** [194] Let \(B_k\) be a binomial tree.

1. \(B_k\) has \(2^k\) nodes.
2. The maximum depth of \(B_k\) is \(k\).
3. There are \(\binom{k}{i}\) nodes at depth \(i\), for \(i = 0, 1, \ldots, k\).
4. In \(B_k\) there is exactly one node that has \(k\) children, which is greater than that of any other node.

**Proof:** The proof is by induction on \(k\). Verifying all properties for \(B_0\) is trivial. We assume all properties hold for \(B_k\) (\(k \geq 0\)).

1. Binomial tree \(B_{k+1}\) consists of two copies of \(B_k\), so \(B_{k+1}\) has \(2^k + 2^k = 2^{k+1}\) nodes.
2. Since by definition \( B_{k+1} \) is built by linking two copies of \( B_k \), such that the root of one is the leftmost child of the root of the other, the maximum depth of \( B_{k+1} \) is one higher than the maximum depth of \( B_k \), which is by our induction hypothesis \( k \). Thus, the maximum depth of \( B_{k+1} \) is \( k + 1 \).

3. Since \( B_{k+1} \) is built by linking two copies of \( B_k \), a node at depth \( i \) in \( B_k \) appears in \( B_{k+1} \) once at depth \( i \) and once at depth \( i + 1 \). Thus, the number of nodes at depth \( i \) in \( B_{k+1} \) is the number of nodes at depth \( i \) in \( B_k \) plus the number of nodes at depth \( i - 1 \) in \( B_k \). From the induction hypothesis, the number of nodes at depth \( i \) in \( B_{k+1} \) is:

\[
\binom{k}{i} + \binom{k}{i-1} = \frac{k!}{i!(k-i)!} + \frac{k!}{(i-1)!(k-i+1)!} = \frac{k!(k-i+1) + k!i}{i!(k-i+1)!} = \binom{k+1}{i}.
\]

4. The only node with greater degree in \( B_{k+1} \) than in \( B_k \) is the root, which has one more child than in \( B_k \). Since the root of \( B_k \) has degree \( k \) (by the induction hypothesis), the root of \( B_{k+1} \) has degree \( k + 1 \).

\[\square\]

**Proposition B.10** The maximum degree of any node in a binomial tree containing \( n \) nodes, is \( \log_2 n \).

**Proof:** This follows directly from property (1) and (4) of Proposition B.9. \[\square\]
Binomial trees can be used for representing sets whose number \( n \) of elements is not always a power of two, by considering the binary decomposition of \( n \).

**Definition B.11** Let \( n = \sum_{i=0}^{\infty} b_i 2^i \) with \( b_i \in \{0, 1\} \). A binomial forest \( F_n \) of order \( n \) is a finite set of binomial trees \( B_i \), where \( b_i = 1 \) in the binary decomposition of \( n \).

**Definition B.12** A binomial heap (or binomial queue) \( H \) is a binomial forest \( F_n \) that satisfies the following properties:

1. Each tree maintains the heap order invariant (see Definition B.5).
2. There is at most one tree in the forest whose root has a given degree.

As mentioned before, the \( \text{Union}(H', H'') \) operation is important on binomial heaps. Let us first consider the trivial case where \( H' = F_{n'} \), \( H'' = F_{n''} \) and \( n' = n'' = 2^k \) for a certain \( k \in \mathbb{N} \), \( k \geq 1 \). Now, both binomial forests \( F_{n'} \) and \( F_{n''} \) consist of one binomial tree \( B_k \). The union of both forests is the binomial tree \( B_{k+1} \) where the original trees \( B_k \) are linked such that the one with the highest key associated with the root node, is now the leftmost child of the other. Thus, the heap order invariant still holds. This type of union is called coupling. The general case, where \( n' \) and \( n'' \) are arbitrarily chosen natural numbers, consists of a sequence of such couplings. Algorithm B.2 describes the general procedure. The algorithm maintains a local forest \( F \) that contains three trees at most. Initially, the forest \( F \) is empty. The new binomial heap is built iteratively by iterating \( i \) from the lowest bit to the highest bit of the binary decompositions of both \( n' \) and \( n'' \). If \( F_{n'} \) contains tree \( B'_i \), this tree is added to the local forest \( F \). If \( F_{n''} \) contains tree \( B''_i \), this tree is also added to the local forest \( F \). Now, we distinguish four situations:

1. The local forest \( F = \emptyset \). Nothing is done in this iteration.
2. The local forest \( F \) contains exactly one tree. This tree is added to the resulting heap \( H \) and \( F \leftarrow \emptyset \).
3. The local forest \( F \) contains exactly two trees. Now, both trees are coupled to a new tree \( B_{i+1} \) and \( F \leftarrow \{B_{i+1}\} \).
4. The local forest \( F \) contains three trees. In this case, one of the trees is chosen arbitrarily to be added to the resulting heap \( H \). The other two are coupled to a new tree \( B_{i+1} \) and \( F \leftarrow \{B_{i+1}\} \).

Clearly, all operations can be done in \( O(1) \) time. The number of iterations is bounded by \( \max\{k', k''\} + 1 \). Since \( k' = O(\log n') \) and \( k'' = O(\log n'') \), the whole procedure takes \( O(\log n) \) time, where \( n = n' + n'' \).

The \( \text{Create} \) operation takes \( O(1) \) time, since only an empty binomial forest has to be returned. The \( \text{Insert}(v, k) \) operation consists of creating a new binomial heap \( H' \), which contains only one binomial tree \( B_0 \) consisting of node \( v \) only. This new heap is then merged into \( H \) by taking \( H \leftarrow \text{Union}(H, H') \). The \( \text{Insert} \) operation therefore takes \( O(\log |V|) \) time. The \( \text{Delete-Min}(v) \) operation consists of scanning all roots in the binomial forest to determine the node with a minimum label. Since
Algorithm B.2: Binomial-Heap-Union(\(H', H''\))

1. \(H \leftarrow \text{Create}\)
2. Let \(n' = \sum_{i=0}^{k'} b'_i 2^i\) be the binary decomposition of \(n'\)
3. Let \(n'' = \sum_{i=0}^{k''} b''_i 2^i\) be the binary decomposition of \(n''\)
4. \(F \leftarrow \emptyset\)
5. for \(i \leftarrow 0\) to \(\max\{k', k''\} + 1\) do
   6. if \(i \leq k'\) and \(b'_i = 1\) then
      7. \(F \leftarrow F \cup B'_i\)
   8. if \(i \leq k''\) and \(b''_i = 1\) then
      9. \(F \leftarrow F \cup B''_i\)
   10. if \(|F| = 1\) then
       11. \(H \leftarrow H \cup F\)
       12. \(F \leftarrow \emptyset\)
   13. else
       14. if \(|F| = 2\) then
           15. Build \(B_{i+1}\) by coupling both trees from \(F\)
           16. \(F \leftarrow B_{i+1}\)
       17. else if \(|F| = 3\) then
           18. Choose arbitrarily one tree \(B \in F\)
           19. \(H \leftarrow H \cup \{B\}\)
           20. \(F \leftarrow F - \{B\}\)
           21. Build \(B_{i+1}\) by coupling both remaining trees from \(F\)
           22. \(F \leftarrow B_{i+1}\)
   23. return \(H\)
all binomial trees in the binomial heap obey the heap order invariant, the node with a minimum label is a root node. If that root node belongs to tree $B_k$, this tree is removed from the current forest. All subtrees of $B_k$ (that actually are the trees $B_0, B_1, \ldots, B_{k-1}$) are added to a new binomial heap $H'$. Now, this new heap is merged into $H$ by taking $H \leftarrow \text{Union}(H, H')$. The whole process takes $O(\log |V|)$ time, since the number of trees is bounded by $O(\log |V|)$ and the UNION procedure takes $O(\log |V|)$ time as well. Finally, the DECREASE-KEY($v, k$) operation takes $O(\log |V|)$, since it is implemented exactly like in a $d-\text{Heap}$ implementation by using a SIFTUP subroutine. From Proposition B.9, we know that the maximum depth of a binomial tree is $O(\log |V|)$.

**Proposition B.13** Algorithm B.1, where the priority queue is implemented with a binomial heap data structure, runs in $O(|V| \log |V| + |A| \log |V|)$ time.

**Proof:** All priority queue operations run in $O(\log |V|)$ time. The number of DELETE-MIN and DECREASE-KEY executions is bounded by $O(|V|)$ and the number of DELETE-MIN executions is bounded by $O(|A|)$. The algorithm therefore runs in $O((|V| + |A|) \log |V|)$ time. \hfill $\square$

It should be noted that the binomial heap is to be preferred over the $d$-heap implementation in situations where two heaps have to be merged. In a $d$-heap implementation, it takes $\Theta(n)$ time to implement the UNION($H', H''$) operation where both $H'$ and $H''$ contain $n$ nodes. Since this operation is not used in Algorithm B.1, the binomial heaps are not commonly used in shortest path algorithms. As stated before, our interest in binomial heaps is motivated by the fact that they form a basis for other heaps.

### B.1.3 Fibonacci heap

We follow Knuth [123] to provide some theory on the Fibonacci sequence.

**Definition B.14** The Fibonacci sequence $F_0, F_1, F_2, \ldots$ is defined by the rule that $F_0 = 0$, $F_1 = 1$, and every further term is the sum of the preceding two.

**Proposition B.15** Let $F_n$ be the $n$th term ($n \geq 1$) of the Fibonacci sequence. If $\phi = (1 + \sqrt{5})/2$ (the so-called golden ratio), then $\phi^{n-2} \leq F_n \leq \phi^{n-1}$.

**Proof:** We prove this by induction. If $n = 1$, we have $\phi^{n-2} = \phi^{-1} < 0.7 < F_1 = 1 = \phi^0 = \phi^{n-1}$. For $n = 2$, we have $\phi^{n-2} = \phi^0 = F_2 = 1 < 1.6 < \phi^1 = \phi^{2-1}$. We assume that $\phi^{m-2} \leq F_m \leq \phi^{m-1}$ for $m = 1, 2, \ldots, n$ ($n > 1$). In particular, $\phi^{n-3} \leq F_{n-1} \leq \phi^{n-2}$ and $\phi^{n-2} \leq F_n \leq \phi^{n-1}$. Adding these inequalities, we get

$$\phi^{n-3}(1 + \phi) = \phi^{n-3} + \phi^{n-2} \leq F_{n+1} = F_{n-1} + F_n \leq \phi^{n-2} + \phi^{n-1} = \phi^{n-2}(1 + \phi).$$

Since $\phi = (1 + \sqrt{5})/2$, we have $1 + \phi = \phi^2$. Thus, $\phi^{n-1} \leq F_{n+1} \leq \phi^n$, which completes the proof. \hfill $\square$
For $n \geq 2$, the Fibonacci sequence satisfies:

$$F_{n+2} = \sum_{i=2}^{n} F_i + 2.$$  \hfill (B.1)

**Proof:** We prove this by induction. Obviously, the equality holds for $n = 2$, since $F_4 = F_3 + F_2 = 3 = F_2 + 2$. We assume the equality holds for a certain $n \geq 2$. For $n + 1$, we have:

$$F_{n+3} = F_{n+2} + F_{n+1} = F_{n+1} + \sum_{i=2}^{n} F_i + 2 = \sum_{i=2}^{n+1} F_i + 2.$$  

This completes the induction proof. \hfill $\Box$

While trying to speedup algorithm B.1, Fredman and Tarjan [73] invented the so-called Fibonacci heap (or F-heap for short). A Fibonacci heap is implemented as a collection (forest) of rooted trees. Each rooted tree maintains the heap order invariant (see Definition B.5). Thus, the root (source) of each tree contains a node with minimum label among all nodes in that tree. Figure B.3 [3] shows an example of a Fibonacci-heap. Again, the label $d_v$ is considered to be equal to $v$. In Figure B.3, we also visualize how a Fibonacci-heap is actually stored. The following pointers are used:

- The predecessor of a node is stored. If a node has no predecessor, this pointer is NIL.
- The pointer to one of the successors of a node is stored. If a node has no successors, this pointer is NIL.
- The successors of a node are interconnected with a so-called circular doubly-linked list.
- The sources of the trees are interconnected with a so-called circular doubly-linked list.
- A special pointer MINKEY is used to keep track of the node with the smallest label.

Furthermore, we store the rank of each node in the heap.

**Definition B.17** The rank of a node in the heap is equal to the number of its (direct) successors.

The doubly-linked lists make it possible to delete a node from these lists in $O(1)$ time. The circular linking makes concatenation of two lists possible in $O(1)$ time. The operation CREATE only initializes the heap by setting MINKEY to NIL, which takes $O(1)$ time. The operation FIND-MIN takes $O(1)$ time as well, since MINKEY points to a node with minimum label. The operation INSERT($v, k$) is performed by
creating a new rooted tree containing node \( v \) only. This tree is added to the forest in \( O(1) \) time.

Before we discuss the other heap operations in more detail, we first describe two subroutines that are used:

- **Link**\((v, w)\)
  
  If \( d_v \leq d_w \) the arc \((v, w)\), is added to the Fibonacci forest (node \( v \) becomes the predecessor of node \( w \)). If \( d_v > d_w \), arc \((w, v)\) is added to the forest.

- **Cut**\((v)\)
  
  Let \( w \) be predecessor of node \( v \). Arc \((w, v)\) is removed from the Fibonacci forest. Thus, node \( v \) becomes the root of a new tree in the forest.

Both subroutines, **Link** and **Cut**, can be processed in \( O(1) \) time. As we will prove later, the operation **Delete-Min** takes \( O(\log |V|) \) amortized time and the operation **Decrease-Key** takes \( O(1) \) amortized time. First, we describe these operations in more detail:

The operation **Delete-Min** is the most time consuming operation in a Fibonacci heap data structure. Since **minkey** points to a node \( u \) with a minimum label, it is easy to return that node \( u \). Before node \( u \) is actually removed from the forest, its children are added to the list of roots. Thus, each child of node \( u \) becomes the root of a new tree in the forest. So far, only \( O(1) \) time is needed. However, we need to
Consolidate the list of all roots in order to get our desired complexity bounds of the heap operations. This Consolidate step consists of finding any two trees in the forest, whose roots \( v \) and \( w \) have the same rank (see Definition B.17) and then performing the subroutine \( \text{Link}(v, w) \). This step is repeated until no two trees with roots of the same rank can be found. Then, all remaining roots are scanned in order to find a new node with minimum label (the pointer \( \text{minkey} \) is updated). During this scan the doubly-linked list maintaining all roots is rebuilt.

In order to determine pairs of tree roots with the same rank, an array indexed by rank (from zero to the maximum rank) is used. Each array position holds a pointer to a tree root (or to \( \text{nil} \)). After adding the children of node \( u \) to the list of roots, the trees are inserted one by one into the appropriate array positions. If an array position is already occupied, then the subroutine \( \text{Link} \) is executed and an attempt is made to store the new tree (with higher rank) into the array. Eventually, we will find a position in the array that is not occupied yet. After inserting all the roots, the array is scanned to determine the new \( \text{minkey} \), to rebuild the doubly-linked list that maintains all roots and to empty all array positions. The total time to Consolidate (and thus to perform the operation \( \text{Delete-Min} \)) is proportional to the maximum rank of any of the nodes manipulated, plus the number of executions of the \( \text{Link} \) subroutine.

The operation \( \text{Decrease-Key}(v, k) \) consists of the following steps. First, the label \( d_v \) is updated: \( d_v \leftarrow k \). Next, if node \( v \) is not a root of a tree, the label \( d_v \) is compared to the label \( d_u \) of its predecessor \( u \). If \( d_v < d_u \), the subroutine \( \text{Cut}(v) \) is executed and, if necessary, the pointer \( \text{minkey} \) is updated. To get the desired complexity bounds, an extra step is introduced. Each non-root node \( w \) that already lost a successor after becoming a non-root node, is cut as well. Thus, one single \( \text{Cut} \) execution might lead to several cascading cuts. See Figure B.4 [73] for an example. If a node becomes a non-root node, that node is ‘unmarked’. As soon as an unmarked non-root node loses its first child, the node is ‘marked’. If a marked non-root node loses a child, the node itself is cut as well. In Figure B.4 marked nodes are colored black.

The cascading cuts are introduced in order to ensure that any non-root node has lost at most one successor. This property is used to prove the following proposition.

**Proposition B.18** [73] *Let \( v \) be any node in a Fibonacci heap. Arrange the successors of node \( v \) in the order they were linked to \( v \), from earliest to latest. Then, the \( i^{th} \) successor of \( v \) has a rank of at least \( i - 2 \).*

**Proof:** Let node \( w \) be the \( i^{th} \) successor of node \( v \), and consider the moment when \( w \) was linked to \( v \). Node \( v \) had at least \( i - 1 \) children at that time (some of which it may have lost after the linking). Directly before the linking, node \( v \) and node \( w \) had the same rank. Thus, node \( w \) had at least rank \( i - 1 \). After the linking, node \( w \) could have lost at most one more child without causing node \( w \) to be cut as a successor of node \( v \). Thus, the rank of node \( w \) is at least \( i - 2 \). \( \square \)
Proposition B.19 [73] A node of rank \( k \) in a Fibonacci heap has at least \( F_k + 2 \) descendants, including itself, where \( F_k \) is the \( k \)th term in the Fibonacci sequence.

Proof: Let \( S_k \) be the minimum possible number of descendants of node with rank \( k \). Obviously, \( S_0 = 1 \), and \( S_1 = 2 \). From Proposition B.18, we have for \( k \geq 2 \):

\[
S_k \geq \sum_{i=0}^{k-2} S_i + 2. \tag{B.2}
\]

If we compare (B.1) and (B.2), it is easy to prove by induction on \( k \) that \( S_k \geq F_{k+2} \).

The trivial cases are \( k = 0 \) where \( S_0 = 1 = F_2 \), and \( k = 1 \) where \( S_1 = 2 = F_3 \). Now, for \( k = 2 \) we have \( S_2 \geq S_0 + 2 = 3 = F_3 \), and for \( k = 3 \) we have \( S_3 \geq S_0 + S_1 + 2 = 5 = F_5 \). Let us assume that \( \sum_{i=0}^{k-2} S_i + 2 \geq F_{k+2} \) for all \( m = 2, 3, \ldots, k \), with \( k \geq 3 \).

Thus, \( S_{k+1} \geq \sum_{i=0}^{k-1} S_i + 2 = S_{k-1} + \sum_{i=0}^{k-2} S_i + 2 \geq S_{k-1} + F_{k+2} \geq \sum_{i=0}^{k-3} S_i + F_{k+2} \geq F_{k+1} + F_{k+2} = F_{k+3} \). This completes the induction proof.

Proposition B.20 The maximum rank of any node in a Fibonacci heap that contains \( n \) nodes is \( \log_\phi n \), where \( \phi \) is the golden ratio (see Proposition B.15).

Proof: Let \( k \) be the maximum rank of all nodes in the Fibonacci heap. From Proposition B.19, we know that the subtree of a node with that rank contains at least \( F_{k+2} \) nodes. There might be more rooted trees in the heap, thus \( n \geq F_{k+2} \).

From Proposition B.15, we have \( F_{k+2} \geq \phi^k \). Hence, \( k \leq \log_\phi n \).

Now, we will provide bounds on the number of times that the subroutines Link and Cut are called. The Cut subroutine is called only during the execution of
a **Decrease-Key** operation. If the label of a non-root node \( v \) is decreased it is checked whether or not the heap order invariant is violated. If the invariant is violated, the node \( v \) is cut from its parent \( u \). We will call this type of cuts ‘actual cuts’ to distinguish them from the ‘cascading cuts’ that might happen as well, due to the fact that marked nodes lose another child. Since the **Decrease-Key** operation is performed \( O(|A|) \) times, the number of actual cuts is bounded by \( O(|A|) \) as well.

**Proposition B.21** [3] While manipulating a Fibonacci heap, the number of cascading cuts is less than, or equal to, the number of actual cuts.

**Proof:** Let \( n_v \) represent the number of successors that node \( v \) has lost after it became a non-root node. For a root node \( u \), \( n_u = 0 \). We use the amortized analysis as described in Section 2.1.7, using a potential function \( \Phi \), defined by:

\[
\Phi = \sum_{v \in Q} n_v
\]

Suppose that a \( \text{Cut}(v) \) is executed and that \( u \) is the parent of \( v \). This operation sets \( n_v = 0 \) and increases the number \( n_u \) by one if \( u \) is a non-root node. If the cut operation was an ‘actual cut’, we had either \( n_v = 0 \) or \( n_v = 1 \) just before the cut. If the cut was a ‘cascading cut’, we had \( n_v = 2 \). Therefore, an actual cut increases the value of \( \Phi \) by at most one, and a cascading cut decreases the value of \( \Phi \) by at least one. If we start with \( \Phi = 0 \), the total decrease in the potential function is bounded by the total increase. Therefore, the number of cascading cuts never exceeds the number of actual cuts.

The total time for all executions of the **Decrease-Key** operation in Algorithm B.1 is bounded by \( O(|A|) \), since each individual \( \text{Cut} \) takes only \( O(1) \) time. Since the **Decrease-Key** is executed \( O(|A|) \) times, we say that the **Decrease-Key** operation runs in \( O(1) \) amortized time.

**Proposition B.22** [3] While manipulating a Fibonacci heap, the number of links is at most \( |V| \) plus the number of cuts.

**Proof:** Consider a potential function \( \Phi \), defined as the number of rooted trees in the forest. Each link operation decreases \( \Phi \) by one and each cut operation increases \( \Phi \) by one. The total decrease in \( \Phi \) is bounded by its initial value plus the total increase in \( \Phi \). The initial value is bounded by \( |V| \) (each node is seen as a separate tree). Thus, the total number of link operations is bounded by \( |V| \) plus the number of cuts.

Since the total number of cuts is bounded by \( O(|A|) \), the total number of links is bounded by \( O(|V| + |A|) \). As we have seen before, the operation **DELETE-MIN** takes \( O(1) \) time plus the time needed to **CONSOLIDATE** the list of roots. Rebuilding the root list takes \( O(\log |V|) \) time, since after linking all roots with the same rank, the
number of trees is bounded by \( O(\log \phi |V|) = O(\log |V|) \) (see Proposition B.20). The total time needed for all \(|V|\) executions of \textsc{Delete-Min} in Algorithm B.1 is bounded by \( O(|V| \log |V| + |V| + |A|) \). The amortized time for a \textsc{Decrease-Key} operation is \( O(\log |V|) \), if we assume \(|A| = O(|V|)\). This assumption is exactly the basis for all data structures described in this chapter.

**Proposition B.23** Algorithm B.1, where the priority queue is implemented with a Fibonacci heap data structure, runs in \( O(|V| \log |V| + |A|) \) time.

**Proof:** As we saw earlier, the operations \textsc{Create} and \textsc{Insert} take \( O(1) \) time. The operation \textsc{Decrease-Key} runs in \( O(1) \) amortized time and the operation \textsc{Delete-Min} runs in \( O(\log |V|) \) amortized time. Thus, Algorithm B.1, using a Fibonacci heap data structure, runs in \( O(|V| \log |V| + |A|) \) time.

In several descriptions of the Fibonacci heap (e.g. [3, 35]) the following invariant is used: no two root nodes have the same rank. This invariant does not influence the complexity bound described above. The \textsc{Consolidate} routine that links all root nodes with the same rank (and rebuilds the root list) has to be executed at the end of the \textsc{Insert} and \textsc{Decrease-Key} operation as well. We believe the original description by Fredman and Tarjan [73], where the \textsc{Consolidate} routine is executed only at the end of each \textsc{Delete-Min} operation, to be more effective.

Clearly, the Fibonacci heap is a breakthrough in the research on priority queue implementations, since it takes \( O(1) \) amortized time to execute any priority queue operations, except for the \textsc{Delete-Min} operation, which takes \( O(\log |V|) \) amortized time. The drawback of Fibonacci heaps is that they are complicated compared to the heaps mentioned earlier. In practice, the Fibonacci heap is often outperformed by other, theoretically less efficient, data structures [23, 35]. We empirically verified this behavior [121]. We strongly advice to use not only a Fibonacci heap in experiments, but also some \( d \)-heap implementations, especially in sparse graphs.

Since the invention of the Fibonacci heap, several other heaps have been presented, trying to match the amortized time complexity of the original Fibonacci heap while being simpler to implement and thus more efficient in practice. Based on amortized analysis, several other heaps are proposed. Sleator and Tarjan [181] introduced the skew heap, an amortized version of the leftist heap. The skew heap achieves the same complexity as the Fibonacci heap, except for \textsc{Decrease-Key}, which takes \( O(\log |V|) \) amortized time. The pairing heap of Fredman et al. [72], an amortized version of the binomial heap, also achieves the same complexity as the Fibonacci heap, except for \textsc{Decrease-Key}, which takes \( O(\log |V|) \) amortized time. The \textit{pairing} heap of Fredman et al. [72], an amortized version of the binomial heap, also achieves the same complexity as the Fibonacci heap, except for \textsc{Decrease-Key} operation for which the amortized time is \( 2^{O(\sqrt{\log \log |V|})} \) [151]. Elmasry [64] provided a variant of the pairing heap that needs only \( O(\log \log |V|) \) amortized time for the \textsc{Decrease-Key} operation. Since the amortized bounds of these heaps are less efficient than the amortized bounds of the Fibonacci heap, we do not discuss these heaps in detail. Recently, Haeupler et al. [101] introduced the Rank-Pairing heap that combines the amortized time complexity of the Fibonacci heap with the simplicity of pairing heaps.
B.1.4 Relaxed heap

The relaxed heap data structure, as proposed by Driscoll et al. [62], achieves the same amortized bounds as the Fibonacci heap (see Section B.1.3). They also described a variant that achieves the same bounds: $O(1)$ time for `DELETE-MIN` and $O(\log |V|)$ for `DELETE-MIN` time, in worst case. The relaxed heap is based on the binomial trees (see Section B.1.2). In a binomial heap data structure, it seems difficult to execute `DECREASE-KEY` in $O(1)$ time, while keeping the heap order invariant. The idea behind relaxed heaps is to allow some violations of the heap order.

**Definition B.24** Let $B_k$ be a binomial tree. A node $v \in B_k$ is good if it is the root, or if its parent $u$ satisfies $d_u \leq d_v$. Otherwise ($d_u > d_v$) node $v$ is bad.

In a binomial heap, each tree obeys the heap order invariant and thus all nodes are good.

In a relaxed heap, bad children ‘play together’ [180] in such way that at least one bad child becomes good. A variety of transformations is applied in order to organize this play, and as a result to decrease the number of bad children in $O(1)$ time.

Each time the `DECREASE-KEY(v,k)` operation is executed, node $v$ is considered as active, since it represents a possible violation of the heap order. Node $v$ can be good or bad. An active node that is bad, might become good by either changing its parent or by changing the label of its parent. A rank relaxed heap allows one active node for each possible rank (see Definition B.17).

**Definition B.25** A rank relaxed heap $\mathcal{H}$ is a binomial forest, which satisfies the following properties:

1. For any $r$, there is at most one active node of rank $r$.
2. Any active node is a first child.
3. There is at most one tree in the forest whose root has a given degree.

Let $\alpha$ denote the number of active nodes in the rank relaxed heap at any point during the execution of Algorithm B.1. Initially, $\alpha = 0$. The `DECREASE-KEY(v,k)` operation sets $d_v \leftarrow k$ and $\alpha \leftarrow \alpha + 1$. Each time a second active node appears at a given rank, one or more transformations are applied to correct the structure. Each transformation either:

(i) decreases $\alpha$, or
(ii) does not change $\alpha$ and does not require further transformations to be executed.

Any sequence of $n$ `DECREASE-KEY` operations, takes $O(n)$ time. There are at most $n$ transformations of type (ii), since they are finishing their sequence of transformations. There are at most $n$ type one transformations, since $\alpha \geq 0$ and $\alpha$ is only increased by a `DECREASE-KEY` operation. Thus, the `DECREASE-KEY` operation
takes $O(1)$ amortized time. The first transform we introduce is the check whether or not node $v$ is bad. If node $v$ is good, node $v$ becomes inactive immediately. Formally, this is a type (i) transformation. We will now describe in more detail the possible transformations that arise when node $v$ is actually a bad node.

### B.1.4.1 Pair transformation

The first transformation is a pair transformation, see Figure B.5. Suppose node $v$ of rank $r$ becomes active and node $v$ is the first child of node $p$ and a grandchild of node $g$ (node $g$ might be $\text{Nil}$, which indicates that $p$ is a root). In case there exists another active node $v'$ with rank $r$, which is the first child of node $p'$ and a grandchild of node $g'$, the pair transformation consists of the following steps:

1. Nodes $v$, $v'$, $p$ and $p'$ are cut from their respective parents. These four nodes now have rank $r$.
2. Without loss of generality, assume $d_p \leq d_{p'}$. Add $p'$ as the first child of node $p$. Hence, node $p$ remains a rank $r + 1$ node.
3. Build the binomial tree $B_{r+1}$ by linking nodes $v$ and $v'$ in the usual way. Note that the one with the largest label, now becomes good and inactive.
4. If $g' \neq \text{Nil}$, then add the root of the new formed binomial tree $B_{r+1}$ to the grandparent $g'$ in its according position.

Now, we distinguish three situations. If the root of the (temporary) newly formed tree $B_{r+1}$ has a label greater than or equal to $d_{g'}$, the root node of $B_{r+1}$ becomes good (and inactive) as well. If however this root node stays inactive, it is either the first child of $g'$ or not. If it is not the first child of $g'$, another transformation has to be executed to restore the properties mentioned in Definition B.25.

Clearly, the pair transformation is a type (i) transformation, since two initially inactive nodes are replaced by at most one inactive node.

---

**Figure B.5:** The pair transformation. White nodes are active. Note that only a part of the heap is displayed. Furthermore, in this picture it is assumed that $d_v \leq d_{v'}$. If $d_v > d_{v'}$, nodes $v$ and $v'$ would appear in each others places in the rightmost tree. The triangles are binomial trees. Instead of writing $B_{r+1}$, we simply write $r$ to denote a binomial tree of rank $r$. 
B.1.4.2 Active sibling transformation

The active sibling transformation is performed on a node \( v \) that becomes active while it is not the leftmost child of its parent \( p \). Furthermore, the left sibling \( s \) of node \( v \) is active as well. Since only a leftmost child can be active, node \( v \) is the second child of \( p \). If node \( s \) is actually good, its status is changed to inactive and instead of an active sibling transformation, a good sibling transformation or a cleaning transformation is performed, which we will describe later.

The active sibling transformation consists of the following steps (see Figure B.6 for an illustration):

1. Both active nodes \( s \) and \( v \) are removed from \( p \). Now, node \( p \) has rank \( r \).
2. If node \( p \) is not a root, node \( p \) is removed from its own parent \( g \).
3. A binomial tree \( B_{r+1} \) is built by linking nodes \( v \) and \( p \). Clearly, node \( v \) is the root node of this tree.
4. A binomial tree \( B_{r+2} \) is built by linking the tree \( B_{r+1} \) with node \( s \).
5. If node \( g \not= \text{NIL} \), then the root of the newly formed binomial tree \( B_{r+2} \) is added to node \( g \) in its according position.

The active sibling transformation is a type (i) transformation, since two active nodes are replaced by at most one. If the root \( c \) of the newly formed binomial tree \( B_{r+2} \) is a bad node, another transformation might be needed to restore the heap properties. If node \( c \) is not the leftmost child, then another active or good sibling transformation is performed. If node \( c \) is the leftmost child and there is another active node with rank \( r+2 \), then a pair transformation is executed.

![Figure B.6: The active sibling transformation. White nodes are active. Note that only a part of the heap is displayed. In this picture, it is assumed that \( d_v \leq d_s \) and \( d_s > d_p \) (since node \( s \) is white in the rightmost tree).](image)

B.1.4.3 Good sibling transformation

Now, suppose node \( v \) is bad and not the leftmost child of its parent \( p \). Furthermore, the left sibling \( s \) of node \( v \) is good. We distinguish two situations:
Figure B.7: The good sibling transformation. White nodes are active. Note that only a part of the heap is displayed. In this picture it is assumed that $d_c \leq d_v$.

1. If the leftmost child $c$ of node $s$ is good, then nodes $c$ and $v$ are swapped (like in the cleaning operation). If node $v$ is the only active node with rank $r$, the sequence of transformations stops, otherwise a pair transformation is executed.

2. If the leftmost child $c$ of node $s$ is bad, nodes $c$, $v$ and $s$ are cut from their respective parents. Node $s$ (now being a $B_r$ tree) is added to node $p$ at the old position of node $v$. Nodes $v$ and $c$ are linked to a new $B_{r+1}$ tree, which is added to node $p$ as well (Figure B.7 illustrates these steps). Note that either node $c$ or node $v$ is still bad, but the other one is now good. Another transformation of type (i) might be needed if the still active node is not the leftmost child of $p$.

B.1.4.4 Cleaning transformation

If node $v$ is the only bad child of rank $r$ and it is not the leftmost child of its parent $p$, while its left sibling $s$ and the leftmost child $c$ of $s$ are good, the cleaning transformation is performed. This transformation is done by simply swapping nodes $v$ and node $c$ (see Figure B.8 for an illustration). Since $d_v < d_p$ and $d_p \leq d_g$, we have $d_v < d_g$. Thus, node $v$ still is a bad active node, but now this node is a leftmost child.

B.1.4.5 Delete-Min

Deleting a node with minimum label, consists of two steps. First, a node $v$ with minimum label has to be determined. Next, node $v$ has to be deleted from $\mathcal{H}$. A node with minimum label is either a root of one of the trees in the forest or an active node. Thus, we have to check $O(\log |V|)$ roots and $O(\log |V|)$ active nodes, in order to determine node $v$.

Removing node $v$ (with rank $r$) requires the following steps. First, the tree with lowest rank in the forest is determined. Let $u$ be the root of this tree. All children of
Figure B.8: The cleaning transformation. White nodes are active. Note that only a part of the heap is displayed.

u are cut from node u, each of them forming a new tree in the forest. If node u had an active child, this child stays active. If u = v, we just drop node v. If node u ≠ v, let p be the parent of node v. For each child c of node v, this node is coupled in increasing order of rank with node u. Either c or u is the root of the new binomial tree. In both cases, u is set to point to the root of this new tree in order to couple this tree with the possible next child c of node v. As soon as all children of node v are coupled this way, node u has rank r and node v is replaced by node u. If node d_u > d_p, node u is made active.

The number of active nodes α cannot increase due to a DELETE-MIN operation. If node u becomes active, node v surely was active. However, α might be decreased by one. Clearly, the execution of DELETE-MIN takes O(log |V|) time.

B.1.4.6 Amortized complexity

In order to prove that the rank relaxed heap has the same amortized complexity as the Fibonacci heap, we still have to prove that the INSERT operation can be performed in O(1) amortized time. Driscoll et al. [62] use a slightly different implementation of Algorithm B.1 to prove that the inserts can be done in amortized constant time. In the initialization phase, all nodes are added to H. Adding a node takes O(1) time, plus the time needed for coupling. Since each node is coupled at most once (each node had at most one parent), the number of couplings is bounded by O(|V|). Thus, the total time needed for all INSERTS takes O(|V|) time, which proves that the INSERT operation takes O(1) amortized time.

However, the constant amortized complexity of the INSERT operation can easily be proven, using our version of Algorithm B.1:

Proposition B.26 Any sequence of n INSERT operations, and k (k ≤ n) DELETE-MIN operations, takes O(n + k log n) time to complete.

Proof: As we have seen before, each DELETE-MIN operation takes O(log n) time to determine the node with smallest key (by checking the roots and the active nodes).
The total number of cuts that are made due to the \texttt{Delete-Min} operations, is bounded by $2k \log_2 n$. Note that both the children of the node with the minimum key and the children of the root with minimum rank, are cut. The number of couplings that are either made during the \texttt{Delete-Min} or during the \texttt{Insert} operation is bounded by $n$, plus the number of cuts. Thus, the number of links is bounded by $n + 2k \log_2 n$. The whole sequence therefore takes $O(n + k \log n)$ time. 

**Proposition B.27** Any sequence of $n$ \texttt{Insert} operations, $k$ ($k \leq n$) \texttt{Delete-Min} operations and $m$ \texttt{Decrease-Key} operations, takes $O(n + k \log n + m)$ time to complete.

**Proof:** Based on the potential $\alpha$ (the number of active nodes), we already have seen that any sequence of $m$ \texttt{Decrease-Key} operations takes $O(m)$ time to complete. Clearly, the \texttt{Insert} operation has no influence on $\alpha$. The \texttt{Delete-Min} might actually decrease $\alpha$ by one, which only improves the bound on the number of transformations. The \texttt{Decrease-Key} does not change the structure of the forest, only the structure of the trees in the forest. Thus, there are no extra cuts that might increase the bound on the number of couplings needed in either the \texttt{Insert} or \texttt{Delete-Min} operation. From Proposition B.27 and the $O(m)$ bound on any sequence of \texttt{Decrease-Key} operations, any sequence of $n$ \texttt{Insert} operations, $k$ ($k \leq n$) \texttt{Delete-Min} operations and $m$ \texttt{Decrease-Key} operations takes $O(n + k \log n + m)$ time to complete.

**Proposition B.28** The rank relaxed heap has the same amortized complexity as the Fibonacci heap. Therefore, Algorithm B.1, where the priority queue is implemented with a rank relaxed heap data structure, runs in $O(|V| \log |V| + |A|)$ time.

**Proof:** The \texttt{Create} operation takes $O(1)$ time to initialize an empty forest. Based on Proposition B.27, we might state that the \texttt{Insert} operation takes $O(1)$ amortized time, while the \texttt{Delete-Min} takes $O(\log |V|)$ amortized time and the \texttt{Decrease-Key} operation takes $O(1)$ amortized time. Thus, the rank relaxed heap has the same amortized complexity as the Fibonacci heap. The complexity for Algorithm B.1 follows from Proposition B.23.

### B.2 Bucket data structures

Let $G = (V, A)$ be a digraph with $c_{vw} \in \mathbb{N}_+$. In [4], it is stated that the existence of $O(|V| \log |V| + |A|)$ implementations of Algorithm B.1, using a heap that is suitable for real valued arc costs, suggests that for integral arc costs an implementation of Algorithm B.1 can be found with a complexity of $O(|V| f(C) + |A|)$, where $C = \max_{(v,w) \in A} c_{vw}$, for some function $f$, with $f$ growing as slowly as possible.

A priority queue can be efficiently implemented by a so-called bucket data structure. In this paragraph, several bucket data structures are described. Dial’s method originally has $f(C) = C$. A variant of Dial’s algorithm has $f(C) = \sqrt{C}$. Several other
implementations result in \( f(C) = \sqrt{C} \), too. The \( k \)-level bucket implementation has \( f_k(C) = k + \sqrt{C} \) and the two-level radix heap has \( f(C) = \log_2 C / \log \log C \).

### B.2.1 Dial’s method

The bucket data structure of Dial [57] can be used to implement a priority queue. The idea is to maintain \( C + 1 \) buckets in a circular way, where \( C = \max_{(v,w) \in A} c_{vw} \). In Figure B.9 [3] such a bucket structure is visualized.

![Circular bucket structure (Dial)](image)

The idea is that bucket \( b \) stores all nodes with \( d_v = b \mod C + 1 \). When a distance label of a node changes, the node is removed from the bucket corresponding to its old label (if the old label was finite) and removed to the bucket corresponding to the new label.

Any node \( v \) can be stored in at most one bucket. Each bucket \( b \) contains a pointer to the first node in the bucket. For any node two pointers are maintained: one pointing to the preceding node in the same bucket and the other pointing to the next node in the same bucket. All these pointers either point to a node or to nil. Thus, the nodes in any bucket are maintained as a so-called doubly-linked list, which makes it possible to add nodes to and remove nodes from a bucket in \( O(1) \) time. The memory needed to store this data structure consists of \( C + 1 + 2|V| \) pointers and an integer \( n \) to denote the so-called active bucket.

This data structure is also called ‘address calculation sort’ because the insertion and deletion of an item from the list simply involves calculating an address in a straightforward manner [58].

The create operation takes \( O(\max(|A|, C)) \) time. It takes \( O(|A|) \) time to determine the maximum arc cost \( C \). The buckets \( 0, 1, \ldots, C \) have to be initialized as...
empty, which takes $O(C)$ time. Furthermore, the active bucket number $n$ is initialized to $n \rightarrow 0$.

Operation $\text{Insert}(v,k)$ takes $O(1)$ time. First, we determine the bucket $b$ where node $v$ should be inserted: $b \leftarrow d_v \mod C + 1$. Node $v$ can be added to bucket $b$ in $O(1)$ time. The operations $\text{Find-Min}$ and $\text{Delete-Min}$ can be performed in $O(C)$ time. Suppose bucket $n$ is empty. We then have to scan for the first non-empty bucket, by repeatedly increasing $n$ (and reset $n$ to 0, once $n = C + 2$). Deleting a node from the first non-empty bucket can be done in $O(1)$ time.

Finally, the operation $\text{Decrease-Key}(v,k)$ takes $O(1)$ time. First, we determine the bucket $b$ where node $v$ currently belongs to: $b \leftarrow d_v \mod C + 1$. Node $v$ is removed from bucket $b$ in $O(1)$ time. Next, we determine the new bucket where node $v$ should be positioned $b \leftarrow k \mod C + 1$. Adding node $v$ to this bucket again takes $O(1)$ time.

**Proposition B.29** The running time of Algorithm B.1, using Dial’s bucket data structure, is $O(C|\mathcal{V}| + |\mathcal{A}|)$.

**Proof:** The execution of operation $\text{Create}$ takes $O(|\mathcal{A}| + C)$ time. The operations $\text{Insert}$ and $\text{Delete-Min}$ are executed at most $|\mathcal{V}|$ times. The total time needed for these operations is thus bounded by $O(C|\mathcal{V}|)$ time. The operation $\text{Decrease-Key}$ is executed at most $|\mathcal{A}|$ times. Thus, the total running time is $O(C|\mathcal{V}| + |\mathcal{A}|)$. \hfill $\Box$

Wagner [195] provides a comparison between Dijkstra’s algorithm (in its basic form) and an implementation of his own, using a data structure very similar to Dial’s bucket data structure, to determine which one will perform better based on the input characteristics $C, |\mathcal{V}|$ and $|\mathcal{A}|$. His results confirm that Dial’s bucket data structure is to be preferred above Dijkstra’s algorithm if both $|\mathcal{V}| > C$ and $|\mathcal{A}| = O(|\mathcal{V}|)$. These results can easily be verified from Proposition B.29 and Proposition 3.6.

### B.2.2 Dial’s method with an overflow bag

In [29], the following improvement of Dial’s bucket data structure is proposed: The number of buckets is set to $B < C + 1$. Beside the buckets, an overflow bag is used and the actual label value for bucket 0 is stored in $d_{\text{min}}$. Initially, $d_{\text{min}} = 0$. Bucket $b$ ($0 \leq b \leq B - 1$) contains nodes with label $d_{\text{min}} + b$. All other nodes (nodes with a label in the range $[d_{\text{min}} + B, d_{\text{min}} + C]$) are stored in the overflow bag. As soon as the active bucket pointer $n$ is $B$, $n$ is reset to 0 and the value $d_{\text{min}}$ is set to $\min\{d_v | v \in \mathcal{Q}\}$, thus the minimum node label among all nodes in the overflow bag. The overflow bag is unpacked. Nodes with labels in the range $[d_{\text{min}}, d_{\text{min}} + B - 1]$ are removed from the overflow bag and added to the appropriate bucket.

**Proposition B.30** The running time of Algorithm B.1, using Dial’s bucket data structure with an overflow bag, is $O(|\mathcal{V}|(\frac{C}{B} + B) + |\mathcal{A}|)$. 
Proof: The operations CREATE, INSERT and DECREASE-KEY have the same bounds as in the original version of Dial’s bucket data structure. The operations FIND-MIN and DELETE-MIN originally took $O(C)$ time. Each time the operation DELETE-MIN is executed in Algorithm B.1, we might have to check $O(B)$ buckets before we reach a non-empty bucket or we need to unpack the overflow bag. Unpacking an overflow bag consists of two passes. In the first pass, the minimal node label $d_{\text{min}}$ among all nodes in the bag is determined. In the second pass, all nodes with a node label in the range $[d_{\text{min}}, d_{\text{min}} + B - 1]$ are removed from the bag and placed in the appropriate bucket. Suppose node $w$ is placed in the overflow bag. Let $v = \pi(w)$, then $d_v - d_w \leq C$. As soon as the overflow bag is unpacked, node $v$ either leaves the bag or remains in the bag. Node $v$ remains in the overflow bag at most $\lfloor C B \rfloor$ times. The total amount of work for executing $|V|$ times the operation DELETE-MIN in Algorithm B.1 is thus $O(|V|(\frac{C}{B} + B))$. The amortized cost for DELETE-MIN is $O(\frac{C}{B} + B)$ and the running time of Algorithm B.1 is $O(|V|(\frac{C}{B} + B) + |A|)$.

If we take $B = \sqrt{C}$, the running time of Algorithm B.1 using Dial’s bucket data structure with an overflow bag is $O(|V|\sqrt{C} + |A|)$.

B.2.3 Approximate buckets

Denardo and Fox [55] introduced a bucket data structure where bucket $b$ contains nodes with labels in the half open interval $[b\Delta, (b+1)\Delta)$, where $\Delta$ is a parameter called bucket width. If all arc lengths are integral, this interval can be rewritten as $[b\Delta, (b+1)\Delta - 1]$. In [29], such buckets are called approximate buckets. By re-using buckets in a circular way (like in Figure B.9), only $\lceil \frac{C}{\Delta} \rceil + 1$ buckets are needed.

Proposition B.31 [55] Let $G = (V, A)$ be a digraph with positive arc costs. Let $\lambda = \min_{(v,w) \in A} c_{vw}$. If $\Delta \leq \lambda$, the label $d_v$ of any node $v$ in the lowest non-empty bucket equals $d(s, v)$ during the execution of Algorithm B.1 implemented with approximate buckets and bucket width $\Delta$.

Proof: Let bucket $n$ be the first non-empty bucket. All nodes in bucket $n$ have a label in the half open interval $[n\Delta, (n+1)\Delta)$. Furthermore, $\min(d_v | v \in Q) \geq n\Delta$, since the buckets 0, 1, ..., $n - 1$ are empty. For any pivot node $v$ and any arc $(v,w) \in \delta^+(v)$, we have $d_v + c_{vw} \geq n\Delta + \lambda \geq (n+1)\Delta$. Since any node $w$ in bucket $n$ has a label $d_w < (n+1)\Delta$, this label cannot be improved anymore.

If Algorithm B.1 is implemented with approximate buckets and bucket width $\Delta \leq \lambda$, it is not necessary to determine the node $v$ in the active bucket $n$ with the smallest label among all nodes in the bucket. Algorithm B.1 remains a label-setting algorithm, no matter in which order the nodes from bucket $n$ are processed. However, if $\Delta > \lambda$, it is required to repeatedly find a node with the smallest label among all nodes in the active bucket $n$ to remain being a label-setting algorithm. If a pivot node $v$ is arbitrarily selected from the active bucket $n$, it might happen
that some nodes re-enter \( Q \) (actually in bucket \( n \)), resulting in a label-correcting algorithm.

**Proposition B.32** Let \( G = (V,A) \) be a digraph with nonnegative integral arc costs. Algorithm B.1, implemented with approximate buckets and bucket width \( \Delta \), runs in \( O(|A|\Delta + |V|(|\Delta + \frac{C}{\Delta}) \) time.

**Proof:** The execution of operation \textsc{Create} takes \( O(|A| + \frac{C}{\Delta}) \) time for determining \( C \) and initializing all buckets to be empty. The operations \textsc{Insert} and \textsc{Decrease-Key} take \( O(1) \) time. The number of possible re-entries of a node \( v \) in \( Q \) is bounded by \( O(\Delta) \), since each time a node re-enters \( Q \) the node is added to the active bucket \( n \) that which contains \( \Delta \) different label values). Thus, the \textsc{Insert} operation is executed \( O(|V|\Delta) \) times. Anytime a node \( v \) re-enters \( Q \) the next \textsc{Delete-Min} is performed in \( O(1) \), since the active bucket \( n \) is non-empty. At most \( |V| \) times, the operation \textsc{Delete-Min} takes \( O(\frac{C}{\Delta}) \) time for determining the first non-empty bucket. Therefore, the running time for all executions of the \textsc{Delete-Min} operation is bounded by \( O(|V|\Delta + |V|\frac{C}{\Delta}) \). The number of executions of the \textsc{Decrease-Key} operation is bounded by \( O(|A|\Delta) \). The running time of Algorithm B.1 is therefore \( O(|A|\Delta + |V|(|\Delta + \frac{C}{\Delta}) \) time.

**B.2.4 \( k \)-level buckets**

In \[29\], the ideas of the overflow bag (see Section B.2.2) and the approximate buckets (see Section B.2.3) are combined to obtain the so-called double bucket implementation. In the double bucket data structure, two types of buckets are maintained: \textit{low-level} and \textit{high-level}. High-level bucket \( b \) contains nodes with labels in the range \([b\Delta, (b+1)\Delta - 1]\) like in the approximate bucket case. As soon as \( n \) (the active high-level bucket) is updated, it points to the first non-empty high-level bucket. All nodes of this high-level bucket are removed to the corresponding low-level buckets. Furthermore, \( m \) is initialized to point to low-level bucket 0. There are \( \Delta \) low-level buckets and low-level bucket \( i \) (\( i = 0, 1, \ldots, \Delta - 1 \)) contains nodes with label value \( n\Delta + i \). Now, the low-level buckets have to be emptied in the usual way (using the active low-level bucket pointer \( m \)). If a node \( w \) is added to \( Q \), it is inserted in low-level bucket \( d_w - n\Delta \) if \( d_w \leq (n+1)\Delta - 1 \), and in high-level bucket \( \lfloor \frac{d_w}{\Delta} \rfloor \) otherwise. As soon as \( m = \Delta \), all low-level buckets are emptied and the process continues by increasing the active high-level bucket pointer \( n \).

If the high-level buckets are re-used in a circular way (like in Figure B.9), we need only \( \lceil \frac{C+1}{\Delta} \rceil \) high-level buckets and \( \Delta \) low-level buckets to implement the double bucket data structure.

**Proposition B.33** Let \( G = (V,A) \) be a digraph with nonnegative integral arc costs. Algorithm B.1, implemented with the double bucket data structure and bucket width \( \Delta \), runs in \( O(|A| + |V|(|\Delta + \frac{C}{\Delta}) \) time.
Proof: The execution of operation \texttt{Create} takes $O(|\mathcal{A}| + \frac{\Delta}{2})$ time for determining $C$ and initializing all buckets to be empty. The operations \texttt{Insert} and \texttt{Decrease-Key} take $O(1)$ time. The operation \texttt{Insert} is executed $O(|\mathcal{V}|)$ times and the operation \texttt{Decrease-Key} is executed $O(|\mathcal{A}|)$ times. The operation \texttt{Delete-Min} is executed $O(|\mathcal{V}|(\Delta + \frac{\Delta}{2}))$ times. If all low-level buckets are empty, it takes $O(\Delta)$ time to determine the first non-empty high-level bucket. If there are non-empty low-level buckets, it takes $O(\Delta)$ time to determine the first non-empty low-level bucket. Furthermore, any node is transferred at most once from a high-level bucket to a low-level bucket. The total time needed for all executions of the \texttt{Delete-Min} operation, is $O(|\mathcal{A}| + |\mathcal{V}|(\Delta + \frac{\Delta}{2}))$. Hence, the amortized cost of the \texttt{Delete-Min} operation is $O(\Delta + \frac{\Delta}{2})$.

From Proposition B.33, it follows that if $\Delta = \Theta(\sqrt{C})$, then the running time of Algorithm B.1 implemented with the double bucket data structure is $O(|\mathcal{A}| + |\mathcal{V}|\sqrt{C})$.

The concept of the double bucket implementation (using two levels of buckets) can easily be extended to a $k$-level ($k \geq 2$) bucket implementation [29, 55, 94]. There are $B = \left\lceil \sqrt[k]{C} \right\rceil$ buckets at each level. Associated to each level $i$ ($i = 0, 1, \ldots, k-1$) are both a base label value $D_i$ and a pointer $n_i$ (towards the currently active bucket). Bucket $b$ ($b = 0, 1, \ldots, B-1$) at level $i$ contains nodes with a label value in the interval $[D_i + bB^i, D_i + (b+1)B^i - 1]$. The base distances are such that:

\[
\begin{align*}
D_{k-1} & = 0 \mod B^k, \\
D_{i-1} & = D_i + n_iB^i, \quad i = 1, 2, \ldots k-1.
\end{align*}
\]

Figure B.10 shows a 4-level bucket data structure.

![Figure B.10: A $k$-level bucket data structure. Here, $C = 128$ and $k = 4$. The grey buckets at each level are already emptied.](image-url)
Each time the label $d_w$ of node $w$ is updated and $w \notin Q$, it is determined at which level $i$ node $w$ has to be inserted:

$$i \leftarrow \min \{j | 0 \leq j \leq k - 1 \land d_w < D_j + B_j + 1\}.$$

A pivot node is always selected from the first non-empty bucket at level 0. Now, suppose all buckets in level 0 are empty. Set $i \leftarrow 1$ and check whether or not there is a non-empty bucket at level $i$ (by repeatedly increasing the active bucket pointer $n_i$, until $n_i = B$). If all buckets on level $i$ are empty, $i$ is increased by 1 and the search is continued. Since $Q \neq \emptyset$, a non-empty bucket will be found eventually at some level $1 \leq i \leq k - 1$. Now, the active bucket $n_i$ in level $i$ is expanded: all nodes in the bucket are moved to the according buckets of level $i - 1$ and $n_{i-1}$ is set to point to the first non-empty bucket at level $i - 1$. As long as $i > 1$, $i$ is decreased by 1 and the process of expanding the first non-empty bucket of level $i$ continues. Eventually, this process results in a non-empty bucket $n_0$ at level 0, from which the pivot node is selected.

For each level, we keep track of the first and last non-empty bucket, which can be done in $O(1)$ time.

**Proposition B.34** Let $G = (V, A)$ be a digraph with nonnegative integral arc costs. Algorithm B.1, implemented with the $k$-level bucket data structure, runs in $O(|A| + |V|(k + \sqrt{C}))$ time.

**Proof:** It takes $O(|A|)$ time to determine $C$. There are $k \lceil \sqrt{C} \rceil$ buckets that have to be initialized to be empty. The operation CREATE takes $O(|A| + k \sqrt{C})$ time. The operations INSERT and DECREASE-KEY take $O(1)$ time. The operation INSERT is executed $O(|V|)$ times and the operation DECREASE-KEY is executed $O(|A|)$ times. The operation DELETE-MIN is executed $O(|V|)$ times. Finding the first non-empty bucket might take $O(k + \sqrt{C})$ time. It takes $O(k)$ time to find a level containing non-empty buckets and it takes $O(\sqrt{C})$ time to find the next non-empty bucket in the current level. Furthermore, we might need to (repeatedly) expand a bucket to the buckets from a lower level. Each node is expanded to a lower level bucket at most $k$ times. Thus, the total time needed for all executions of the DELETE-MIN operation, is $O(|V|(k + \sqrt{C}))$ and the amortized cost for the DELETE-MIN operation is $O(k + \sqrt{C})$. The running time of Algorithm B.1, implemented with the $k$-level bucket data, is $O(|A| + k \sqrt{C} + |V|(k + \sqrt{C}))$. Normally, $k < |V|$ resulting in an $O(|A| + |V|(k + \sqrt{C}))$ bound on the running time. \hfill $\square$

### B.2.5 Radix heap

So far, we only considered bucket implementations with a fixed bucket width for all buckets. The so-called **radix-heap** by Ahuja et al. [4] uses variable bucket widths. We follow the explanation of [3]. In a radix heap, $B = 1 + \lceil \log_2 (|V(C)|) \rceil$ buckets are maintained. Bucket 0 has width $\Delta_0 = 0$ and bucket $b$ ($b = 1, 2, \ldots, B - 1$) has width $\Delta_b = 2^{b-1}$. It is important to realize that:
\[ \Delta_b = \sum_{i=0}^{b-1} \Delta_i, \quad 1 \leq b \leq B - 1. \] (B.3)

In a radix heap a range, a closed interval of integers \([\ell_b, u_b]\), is assigned to each bucket \(b\). Initially, \(\ell_0 = u_0 = 0\), \(\ell_b = 2^{b-1}\), and \(u_b = 2^b - 1\) for \(b = 1, 2, \ldots, B - 1\). These ranges will be changed dynamically, but \(u_b - \ell_b\) will always be less than \(\Delta_b\) for all \(b = 0, 1, \ldots, B - 1\). It should be noted that some ranges might become empty during the process.

Every time the active bucket \(n\) points to a bucket with a range such that \(u_n - \ell_n \geq 1\), all nodes in that bucket are examined to determine the minimum label value:
\[ d_{\text{min}} \leftarrow \min \{ d_v | b_v = n \}. \] Since all buckets \(0, 1, \ldots, n - 1\) are empty, these buckets can be re-used. The ranges of the buckets \(0, 1, \ldots, n - 1\) are updated. Updating the ranges of the bucket \(0, 1, \ldots, n - 1\) means redistributing the interval \([d_{\text{min}}, u_n]\) over these buckets as follows:

\[
\begin{cases}
\ell_0 = u_0 = d_{\text{min}} \\
\ell_b = \ell_{b-1} + 1, & b = 1, 2, \ldots n - 1 \\
u_b = \min \{ u_{b-1} + \Delta_b, u_n \}, & b = 1, 2, \ldots n - 1
\end{cases}
\] (B.4)

If \(\ell_b > u_b\) for any bucket \(b = 1, 2, \ldots, n - 1\), the range of bucket \(b\) is set to be the empty set. The range of bucket \(n\) is always set to be the empty set, after redistributing the range of bucket \(n\). All nodes of bucket \(n\) are transferred to the corresponding bucket (lower than \(n\)) and \(n\) is reset to 0. Notice that from (B.3), we are sure that the original range fits in the buckets in the first \(n - 1\) buckets.

The radix heap is certainly not an ‘address calculation sort’ method, like Dial’s bucket data structure. It takes \(O(B)\) time to find the appropriate bucket \(b\) for a node \(v\). The range of bucket \(b\) must contain \(d_v\). To avoid unnecessary searches, we store for each node the bucket \(b_v\) where \(v\) is currently placed. Initially, \(b_v \leftarrow -1\) for all \(v \in |V|\).

**Proposition B.35** Let \(G = (V, A)\) be a digraph with nonnegative integral arc costs. Algorithm B.1 implemented with the radix heap data structure runs in \(O(|A| + |V| \log(|V|C))\) time.

**Proof:** It takes \(O(|A|)\) time to determine \(C\). There are \(O(\log(|V|C))\) buckets that have to be initialized. The operation CREATE takes \(O(|A| + \log(|V|C))\) time. Any node \(v \in V\) might enter any bucket at most once. Thus, the total work for determining the accurate bucket, which is needed during the execution of the operations INSERT, DECREASE-KEY and DELETE-MIN (while redistributing the current bucket range), takes \(O(|V|B) = O(|V| \log(|V|C))\) time. Determining the first non-empty bucket (for the operation DELETE-MIN) takes \(O(B) = O(\log(|V|C))\) time. The operations DELETE-MIN and INSERT are executed \(O(|V|)\) times and the operation DECREASE-KEY is executed \(O(|A|)\) times. Thus, Algorithm B.1, implemented with a radix heap, runs in \(O(|A| + |V| \log(|V|C))\) time. \(\square\)
In the original paper [4], the idea of implementing the buckets in a circular way (see Figure B.9) is used in its basic form. If \( w \in Q \), we have \( d_w \leq d_v + C \), where \( v \) is the pivot node last selected. At any moment during the algorithm, we have:

\[
\max(d_v | v \in Q) - \min(d_v | v \in Q) \leq C. \tag{B.5}
\]

The radix heap is implemented with only \( B = \lceil \log_2(C + 1) \rceil + 2 \) buckets. The bucket widths are initialized as described above, but the size of the last bucket \( B - 1 \) is set to \( |V|C + 1 \). Instead of (B.3), we now have:

\[
\sum_{i=0}^{b-1} \Delta_i \geq \min(\Delta_b, C + 1), \quad 1 \leq b \leq B - 1. \tag{B.6}
\]

From (B.5) and (B.6), it is clear that it is possible to redistribute all nodes of bucket \( n \) over the first \( n - 1 \) buckets by changing the ranges of those buckets in the usual way. If bucket \( B - 1 \) is redistributed, its range is changed to \([\max(d_v | v \in Q) + 1, |V|C + 1]\). Note that the maximum node label in \( Q \) can easily be determined during the transferal of the nodes from bucket \( B - 1 \). While implementing the radix heap, it is not necessary to keep track of the lower bounds of the bucket ranges. They can easily be recalculated from the upper bounds. Searching the bucket with the correct range for a node \( v \) is done by scanning the buckets from right (\( b = B - 1 \)) to left (\( b = 0 \)). As soon as \( d_v \leq u_b \), the appropriate bucket for node \( v \) is found. When during a DECREASE-Key operation a node \( v \) does not fit in the range of bucket \( b_v \) anymore, the scanning is started from bucket \( b_v - 1 \) downwards to bucket 0.

**Proposition B.36** Let \( G = (V, A) \) be a digraph with nonnegative integral arc costs. Algorithm B.1, implemented with the radix heap data structure and using \( B = \lceil \log_2(C + 1) \rceil + 2 \) buckets, runs in \( O(|A| + |V| \log C) \) time.

**Proof:** Similar to the proof of Proposition B.35. The number of buckets is now \( O(\log C) \) instead of \( O(\log(|V|C)) \).

### B.2.6 Two-level radix heap

In [4], a two-level implementation of the radix heap is proposed. Like any other \( k \)-level implementation of a bucket data structure, the idea is that for a reduction of the algorithm’s running time, it is needed to lower the number of re-insertions and thus the amount of buckets (see [55]). In a two-level radix heap implementation, defined by a parameter \( K \), the number of buckets is \( B = \lceil \log_K(C + 1) \rceil + 1 \) and the bucket widths are defined as follows: \( \Delta_b = K^{b+1} \) for \( b = 0, 1, \ldots, B - 2 \), and \( \Delta_{B-1} = |V|C + 1 \). Each bucket \( b \) has an associated range \([\ell_b, u_b]\). Initially, \( u_b = \sum_{i=0}^{b-1} K^{i+1} - 1 \) \((b = 0, 1, \ldots, B - 2)\) and \( u_{B-1} = |V|C + 1 \), \( \ell_0 = 0 \) and \( \ell_b = u_{b-1} + 1 \) \((b = 1, 2, \ldots, B - 1)\).
For \( b = 0, 1, \ldots, B - 2 \), bucket \( b \) is partitioned into \( K \) segments, each of width \( K^b \).

The \( k^{th} \) \((k = 1, 2, \ldots, K)\) segment of bucket \( b \) is referred to by the ordered pair \((b, k)\). The last bucket \( B - 1 \) consists of a single segment. Segment \((b, k)\) has an associated range \([r(b, k - 1) + 1, r(b, k)]\), where \( r(b, k) \) is defined as follows:

\[
r(b, k) = \max\{u_{b-1}, u_b - (K - k)K^b\}.
\]

If \( d_v \in [\ell_b, u_b] \), the value of \( k \) such that \( d_v \in [r(b, k - 1) + 1, r(b, k)] \) can be computed in constant time using the formula:

\[
k = K - \left\lfloor \frac{u_b - d_v}{K^b} \right\rfloor.
\] (B.7)

Each node is stored in the two-level radix heap in the appropriate segment. After determining the appropriate bucket \( b \), the appropriate segment \( k \) is determined using (B.7). The DELETEMIN operation is implemented like in the original radix heap, except that only the nodes of a single segment are redistributed instead of all nodes in the entire bucket. To perform a DELETEMIN operation, the first non-empty bucket \( n \) is determined. Next, the first non-empty segment \( k \) within bucket \( n \) is determined. If \( b = 0 \), any node in segment \((b, k)\) can be selected as a pivot node. If \( b > 0 \), all nodes in segment \((b, k)\) are scanned in order to determine the minimum label \( d_{\text{min}} \) among them. The range of the buckets \( 0, 1, \ldots, b - 1 \) is re-calculated according to (B.4). If \( k < K \), the lower bound of the range of bucket \( b \) is set to \( \ell_b = r(b, k) + 1 \). If \( k = K \), the range of bucket \( b \) is set to be the empty set.

**Proposition B.37** Let \( \mathcal{G} = (V, A) \), be a digraph with nonnegative integral arc costs. Algorithm B.1, implemented with the two-level radix heap data structure and parameter \( K \), runs in \( O(|A| + |V|(B + K)) \) time.

**Proof:** The operation CREATE takes \( O(|A|) \) time to determine \( C \) and \( O(B + BK) \) time to initialize all buckets and segments to be the empty set. The total number of executions of the INSERT and DELETEmIN operations is bounded by \( O(|V|) \). The number of executions of the DECREASE-KEY is bounded by \( O(|A|) \). Each node enters any bucket at most once. The total work for determining the appropriate bucket during the executions of the INSERT, DECREASE-KEY and DELETEmIN operations is thus \( O(|V|B) \). Determining the appropriate segment inside the bucket can be done in constant time using (B.7). During the execution of the DELETEmIN operation, the first non-empty bucket has to be determined. If we use a flag to identify whether or not all segments of a bucket are empty, finding the first non-empty bucket takes \( O(B) \) time. Furthermore, the first non-empty segment has to be determined, which takes \( O(K) \) time. The total running time of Algorithm B.1 is \( O(|A| + |V|(B + K) + BK) \). Commonly, \( K \ll |V| \) and thus, the running time is bounded by \( O(|A| + |V|(B + K)) \). \( \square \)
Choosing $K = \Theta(\log C / \log \log C)$ gives $B = \lceil \log_K (C + 1) \rceil + 1 = O(\log C / \log \log C)$ and thus the running time of Algorithm B.1 implemented with the two-level radix heap runs in $O(|A| + |V| \log C / \log \log C)$ time.

In [4], an extension is made to the two-level radix heap by adding a Fibonacci heap (see Section B.1.3). This heap is used to keep track of the non-empty segments. Each segment $(b, k) (b = 0, 1, \ldots B - 1, k = 1, 2, \ldots K)$ gets a unique index: $Kb + k$. If a node $v$ is added to $Q$, the index of the segment where node $v$ is located, is stored with node $v$. The Fibonacci heap executes the operations Insert and Delete-Min in $O(1)$ amortized time and the operation Delete-Min in $O(\log |V|)$ amortized time.

Ahuja et al. exploited the fact that the possible keys in the Fibonacci heap are $O(KB)$ and $KB << |V|$. They extended the Fibonacci Heap in such a way that the Delete-Min operation is executed in $O(\log \min(|V|, KB))$ amortized time. If $K = 2^{\lceil \sqrt{\log_2 C} \rceil}$, we have $B = O(\log_K C) = O(\sqrt{\log C})$, and $BK = O(\sqrt{\log C})$ as well. Therefore, the running time of Algorithm B.1 with a two-level radix heap, combined with their Fibonacci heap, is $O(|A| + |V| \sqrt{C})$.

B.2.7 Hot queues

Cherkassky et al. [30] exploited the idea to combine $k$-level bucket data structures and heaps. They introduce the heap-on-top queue (or hot queue). When buckets are sparsely occupied, the heap is used instead of the buckets. The hot queue takes advantage of the best performances of both underlying data structures (the heap and the bucket data structure).

B.2.8 Heuristic estimates and bucket data structures

In Section 5.1.4, it is proved that an A* algorithm (see Algorithm 5.1) with a consistent heuristic estimator (see Definition 5.5) is equivalent to Dijkstra’s algorithm using the arc costs $c'_{vw} = c_{vw} + h_w - h_v$. Thus, we can use any bucket data structure in an A* algorithm with a consistent heuristic estimator, taking $C = \max_{(v,u) \in A} c_{vw} + h_w - h_v$. Since it might be expensive to determine $C$ this way, we propose another way to use a bucket data structure in an A* algorithm with a consistent heuristic estimator. Instead of determining the appropriate bucket to store a node $v$ based on its node label $d_v$, we now determine the appropriate bucket based on $d_v + h_v$. Furthermore, we take $C = |V|(\max_{(v,u) \in A} c_{vw}) - h_s$. Especially when $h_s$ is a high lower bound for $d(s, t)$, the number of explored buckets will stay low.

Not all bucket data structures are suitable to use in an A* algorithm with an inconsistent but optimistic (see Definition 5.3) heuristic estimator. The use of an inconsistent heuristic estimator results in a label-correcting algorithm. Any node $v$ might enter $Q$ more than once. A bucket data structure re-using old buckets is not suitable for such an A* algorithm. For example, neither the radix heap can be used in a label-correcting environment, nor can Dial’s implementation re-using the
buckets in a circular way. However, Dial’s bucket data structure can be implemented such that for each possible label value, a separate bucket is defined. Such an implementation contains $C(|V| - 1) + 1$ buckets. Any time a node $v$ enters a bucket $b$, we set the active bucket pointer to $n \leftarrow \min(n, b)$. 
The all-pairs shortest path problem

So far, we mainly focussed on the SSSD-SPP. In this appendix, we will discuss the so-called all-pairs shortest path problem (APSP). All algorithms discussed before, can be used to solve the APSP. Obviously, the APSP can be solved by executing the Generic Shortest Path algorithm (Algorithm 3.4) \(|V|\) times, where each node \(v \in V\) plays the role of source node once. More sophisticated algorithms to solve the APSP exist.

In the SSSD-SPP case, one can try to reduce the number of visited nodes by using heuristic estimators (Chapter 5) and bidirectional algorithms (Chapter 4). In order to solve the APSP however, the SSSD-SPP has to be solved \(|V| (|V| - 1)\) times. These techniques are quite useless for the APSP, since all nodes have to be visited.

This appendix is structured as follows: First, we present some general theory about the APSP in Section C.1. We encountered some test-sets for the APSP where negative arc costs were included. So far, we considered only graphs with positive arc costs. For completeness, we briefly discuss the consequences of negative arc costs in Section C.2. Next, we describe two well-known algorithms to solve the APSP: FLOYD-WARSHALL (Section C.3) and JOHNSON (Section C.4). A recent algorithm SNOWBALL is described in Section C.5. We performed an experimental evaluation of all these algorithms. The results of this evaluation are presented in Section C.6.

C.1 Generic APSP algorithm

First, we present some theory in order to describe the Generic APSP Algorithm. Let \(G = (V,A)\) be a digraph. For each pair of nodes \(v,w \in V\), a distance label \(d_{vw}\) is maintained. If \(d_{vw}\) is finite, it represents the length of a \(v\)-\(w\) path \(P_{vw}\). Thus, \(d_{vw}\) is an upper bound for the shortest path length from node \(v\) to node \(w\). The
main theorem that is used in the Generic APSP algorithm is a generalization of Theorem 2.50:

**Theorem C.1** [3] Let \( G = (V, A) \) be a strongly connected digraph that contains no negative cycles and let \( D = (d_{vw}) \) be a \(|V| \times |V|\) matrix where \( d_{vw} \) \((v, w \in V)\) represents the length of a directed \( v-w \) path (or \( d_{vw} = \infty \)). \( D \) satisfies \( d_{vv} = 0 \) for all \( v \in V \) and \( d_{vw} \leq c_{vw} \) for all \((v, w) \in A\). Each entry \( d_{vw} \) represents the length of a shortest \( v-w \) path if and only if

\[
d_{vw} \leq d_{vu} + d_{uw}, \quad \forall \ u, v, w \in V.
\]

**Proof:** For every pair of nodes \( v, w \), it is necessary that \( d_{vw} \) satisfies (C.1) in order to represent the length of a shortest \( v-w \) path. If \( d_{vw} > d_{vu} + d_{uw} \), the path represented by \( d_{vw} \) can be shortened by the concatenation of the \( v-u \) path (represented by \( d_{vu} \)) and \( u-w \) path (represented by \( d_{uw} \)).

Now, we have to prove that the inequalities (C.1) are sufficient to establish the fact that \( d_{vw} \) represents the length of a shortest \( v-w \) path for any pair of nodes \( v, w \). Let \( P_{vw} = (u_1 = v, (u_1, u_2), u_2, \ldots, (u_{k-1}, u_k), u_k = w) \) be some directed \( v-w \) path. The inequalities (C.1) imply that:

\[
d_{vw} = d_{u_1, u_k} \leq d_{u_1, u_2} + d_{u_2, u_k} \leq c_{u_1, u_2} + d_{u_2, u_k} \\
\leq c_{u_1, u_2} + d_{u_2, u_3} + d_{u_3, u_k} \leq c_{u_1, u_2} + c_{u_2, u_3} + d_{u_3, u_k} \\
\vdots \\
\leq c_{u_1, u_2} + c_{u_2, u_3} + \cdots + c_{u_{k-1}, u_k} = \ell(P_{vw})
\]

Therefore, \( d_{vw} \) is a lower bound on the length of any directed \( v-w \) path. By assumption, \( d_{vw} \) is also an upper bound on the length of the shortest \( v-w \) path. Hence, \( d_{vw} \) equals the length of a shortest \( v-w \) path.

Algorithm C.1 initializes the matrix \( D \) based on a directed graph \( G = (V, A) \). This initialization takes \( O(|V|^2 + |A|) \) time. At the end of the algorithm, we have:

\[
d_{vw} = \begin{cases} 
0 & \text{if } v = w, \\
c_{vw} & \text{if } v \neq w, (v, w) \in A, \\
\infty & \text{otherwise}.
\end{cases}
\]

The Generic APSP algorithm is shown in Algorithm C.2.

**Theorem C.2** Let \( G \) be a strongly connected graph with no negative cycles. Then, the Generic APSP algorithm terminates and upon termination for all \( v, w \in V \), \( d_{vw} \) is the length of a shortest \( v-w \) path.

**Proof:** We first prove that the algorithm will terminate. Each time an entry \( d_{vw} \) is decreased, \( d_{vw} \) equals the length of an \( v-w \) path. The graph \( G \) does not contain
cycles with negative length. Therefore, each time a label \( d_{vw} \) is decreased, the \( v-w \) path is a simple path. The number of different simple \( v-w \) paths is finite. Thus, each label \( d_{vw} \) can only be decreased a finite number of times.

Secondly, we show that for all \( v, w \in V \), upon termination, \( d_{vw} \) is the length of a shortest \( v-w \) path. The inequalities (C.1) hold at the end of the algorithm, while, by initialization (see Algorithm C.1) we have \( d_{vv} = 0 \) for all \( v \in V \) and \( d_{vw} \leq c_{vw} \) for all \( (v, w) \in A \). By Theorem C.1, we know that \( d_{vw} \) equals the length of a shortest \( v-w \) path.

\[ \Box \]

### C.2 Negative arc costs

Although we are mainly interested in graphs with nonnegative arc costs, we perform some experiments on graphs with negative arc costs in this appendix as well. If some arc costs are negative, we might encounter two types of difficulties:

- The graph considered might contain a cycle with negative length.
- Any label-setting algorithm (like Dijkstra’s algorithm) might need an exponential number of iterations.

#### C.2.1 Negative Cycle Detection Problem

If a strongly connected digraph \( G = (V, A) \) contains a cycle of negative length, clearly there exists no shortest path between any two nodes \( v, w \in V \). Determining whether or not a graph contains a negative cycle is called the Negative Cycle Detection Problem (NCDP) in related literature.

The Generic APSP algorithm can be used to solve the NCDP.
Proposition C.3 Let \( G = (V, A) \) be a digraph and \( C = \max_{(u, w) \in A} |c_{uw}| \). The graph \( G \) contains a cycle of negative length if and only if, during the execution of the Generic APSP algorithm in line 3, one of the following situations occurs:

- If \( v = w \) and \( d_{vw} < 0 \).
- If \( v \neq w \) and \( d_{vw} < -|V|C \).

**Proof:** Suppose that at a certain iteration, we have \( v = w \) and \( u \) such that \( d_{vu} < d_{uv} + d_{vw} \). Since \( d_{vu} \) is initialized as \( d_{vu} = 0 \), the label \( d_{vw} \) will be negative after the update. Clearly, \( u \neq v \). Furthermore, \( d_{vu} \) is the length of a \( v-u \) path and \( d_{uw} \) is the length of a \( u-v \) path. The concatenation of these two paths is a cycle (possibly not simple) with negative length. We now consider the second situation. Suppose that, at any iteration, we have \( d_{uv} < -|V|C \). Let \( P_{vw} \) be a corresponding \( v-w \) path. Any simple \( v-w \) path \( P \) has length \( \ell(P) \geq -(|V| - 1)C \), since a simple path contains at most \( |V| - 1 \) arcs. Thus, \( P_{vw} \) must contain at least one cycle of negative length.

If the graph \( G \) contains a negative cycle, then eventually either \( d_{vw} < 0 \) for some \( v \in V \) or \( d_{uw} < -|V|C \) for some pair of nodes \( v, w \in V \), since the iterations will be continued and at each iteration a label \( d_{vw} \) is decreased.

A well-known and commonly used method to solve the NCDP for a strongly connected graph \( G = (V, A) \) is based on Bellman-Ford-Moore (Algorithm 3.7).

Proposition C.4 Let \( G = (V, A) \) be a strongly connected digraph. The graph \( G \) contains a cycle of negative length if and only if, after the execution of Algorithm 3.7 for a randomly chosen source node \( s \in V \), there exists an arc \((v, w) \in A\) for which \( d_w > d_v + c_{vw} \).

**Proof:** At the end of Algorithm 3.7, the labels \( d_v \) represent the length of some shortest \( s-v \) path that contains at most \( |V| - 1 \) arcs. Now suppose there exists an arc \((v, w) \in A\) for which \( d_w > d_v + c_{vw} \). Since \( d_w \) equals the length of a shortest path among all \( s-w \) paths that contain at most \( |V| - 1 \) arcs, obviously there exists a shorter \( s-w \) path via node \( v \) that contains \( |V| \) arcs. Clearly, this is not a simple path. Since \( \ell(P_{sw}) < d_w \), this path contains at least one cycle of negative length.

Suppose on the other hand that \( G \) contains at least one cycle of negative length. Clearly, \( G \) contains at least one simple cycle of negative length. Let \((v = v_0, (v_0, v_1), v_1, \ldots, (v_{k-1}, v_k), v_k = v)\) be a simple cycle with minimal (thus, negative) length. Every time the label \( d_{v_i} \) (\( 0 \leq i \leq k \)) of node \( v_i \) in this cycle is updated, we have \( d_{v_j} < d_{v_j} + c_{v_j, v} \) where \( j = (i + 1) \mod k \). Thus, as soon as \( d_v \) is updated, there will always be an arc \((v, w) \in A\) for which \( d_w > d_v + c_{vw} \). Since the graph \( G \) is strongly connected, a simple \( s-v \) path exists. During the execution of Algorithm 3.7, all simple paths in the graph are considered. Hence, the label of node \( d_{v_0} \) is updated at least once. This completes the proof.

Most of the algorithms to solve the NCDP combine a shortest path algorithm with a cycle-detection algorithm. Cherkassky and Goldberg [28] present an extensive
C.2 Negative arc costs

overview of possible combinations of those two types of algorithms. Especially Tar-ajan’s algorithm, which combines BELLMAN-FORD-MOORE with subtree disassembly [186], turns out to be very efficient. The main idea is that every time the label $d_w$ of a node $w$ is updated, based on an arc $(v, w) \in \mathcal{A}$, it is checked whether or not $v$ is an element of the subtree rooted at node $v$. If so, the graph contains a negative cycle. Furthermore, all nodes in the shortest path subtree rooted at $w$ (except for node $w$ itself) are removed (disassembled) from the candidate set $\mathcal{Q}$. Clearly, all the removed nodes had a non-sharp label at that moment (see Definition 3.28). Therefore, the remaining shortest path algorithm is still correct. Keeping track of the shortest path subtree means that an ADT (see Definition 3.7) has to be implemented, which supports the following operations:

- CREATE($v$). Return a new tree that contains only node $v$.
- ADD($v, w$). Add a node $w$ as a leaf of node $v$.
- REMOVE($v$). Remove a node $v$ and all its descendants from the tree.
- LIST($v$). Return a set of all nodes in the subtree rooted at node $v$.

Lewandowski [130] provides two different implementations of this ADT. A straightforward implementation maintains four pointers for each node, like in a tree of a Fibonacci-heap implementation (recall Figure B.3). One pointer to the parent of the node and another pointer to a child of the node. The children of a node are connected by a circular doubly linked list (the other two pointers therefore point to the previous and next sibling of the node). Clearly, the operations CREATE($v$), ADD($v, w$) and REMOVE($v$) can be performed in $O(1)$ time by appropriate updates of these pointers. The LIST($v$) operation takes $O(n)$ time, where $n$ is the number of nodes in the subtree of node $v$, by a Depth First Search of the subtree rooted at node $v$. However, since a node $w$ can only be found during the LIST($v$) operation if node $w$ is also added by a ADD($u, w$) operation (for any $u \in \mathcal{V}$), the total number of nodes found by all LIST operations is bounded by the total number of ADD operations. Since the total number of ADD operations is bounded by $O(|\mathcal{V}| |\mathcal{A}|)$ (as can readily be seen from Algorithm 3.7), the total time needed for all LIST executions is bounded by $O(|\mathcal{V}| |\mathcal{A}|)$ as well. Thus, the LIST operation runs in $O(1)$ amortized time (see Section 2.1.7). We conclude that the addition of the disassembling of subtrees to BELLMAN-FORD-MOORE does not change the running time of $O(|\mathcal{V}| |\mathcal{A}|)$ for any directed graph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$.

Although originally Tarjan’s algorithm is used to solve the NCDP, the resulting algorithm is highly competitive to the BELLMAN-FORD-MOORE. Cherkassky and Goldberg [28] provide an explanation for this behavior by proving that Tarjan’s algorithm scans only nodes with sharp labels. This follows immediately from the slightly stronger proposition we present here.

**Proposition C.5** Tarjan’s algorithm is globally sharp.

**Proof**: Let $\tilde{\mathcal{G}} = (\tilde{\mathcal{V}}, \tilde{\mathcal{A}})$ with $\tilde{\mathcal{V}} = \{v | d_v < \infty \wedge \pi_v \neq \text{NIL}\}$ and $\tilde{\mathcal{A}} = \{(\pi_v, v) | v \in \tilde{\mathcal{V}} \setminus s\}$ be the shortest path tree rooted at $s$. A globally sharp algorithm is by Definition 3.29
an algorithm for which, at any moment that the shortest path tree is updated, all labels in that tree are sharp. By Definition 3.28, $d_v$ is a sharp label if, for any predecessor $u$ of node $v$ in the tree $\tilde{G}$, the following equality holds: $d_u + \ell(P_{uv}) = d_v$ (where $P_{uv}$ is the (unique) $u$-$v$ path in $\tilde{G}$). A label $d_v$ becomes non-sharp as soon as the label $d_u$ of one of its predecessors $u$ is updated. But in that case, node $v$ belongs to the subtree of node $u$ and is immediately disassembled from $\tilde{G}$ by resetting $\pi_v \leftarrow \text{NIL}$. Thus, every time a label $d_v$ is updated (and thus the structure of $\tilde{G}$ is changed), the resulting shortest path tree $\tilde{G}$ contains only nodes with sharp labels.

Since $Q \subseteq \tilde{V}$, it follows immediately that Tarjan’s algorithm only scans nodes with sharp labels.

### C.2.2 Exponential behavior of Dijkstra’s algorithm

Let $G = (V, A)$ be a digraph with no cycles of negative length, although one or more arc costs might be negative. Dijkstra’s algorithm as originally stated by Dijkstra [59] might fail to find the shortest path tree rooted at node $s \in V$. In the original description, a node $v$ that has been scanned will never re-enter $Q$, since its incoming arcs $\delta^-(v)$ are not considered anymore. Dijkstra’s algorithm will in that case fail to determine the shortest path from node 1 to node 4 in the graph of Figure C.1. After the first iteration, node $d_2 = 2$, $d_3 = 1$ and $d_4 = 0$. Dijkstra will choose node 4 as the next pivot node, no labels will be updated in the second iteration. In the third iteration, node 3 is selected as a pivot node. Again, no labels will be updated. In the fourth iteration, node 2 is selected as a pivot node. The label of node 3 will not be updated (since node 3 is already scanned) and so the original algorithm stops here, the shortest path from node 1 to node 4 (which goes via node 2 and 3) is left undetermined.

![Figure C.1: Example of a graph where Dijkstra’s algorithm (in its original description) fails.](image)

However, if the algorithm is implemented as stated in Algorithm 3.6, it will correctly determine the shortest path tree rooted at $s$. Although it should be noted that not all implementations of a priority queue can be used. There are priority queue implementations explicitly using the fact that the label values of the nodes extracted
from the priority queue, are monotonically non-decreasing. The radix-heap (Section B.2.5) for example re-uses the space initially mentioned for lower label values. Other priority queue implementations (like the $d$-heap and Fibonacci-heap) allow label updates and insertions that are lower than the label of the last pivot node deleted from the priority queue. Furthermore, the possible re-entry of a node $v$ in $Q$ makes it impossible to stop as soon as node $t$ (the destination node) is about to be scanned. Instead of waiting for $Q$ to be empty, we can still use the improved stop condition from any label-correcting algorithm (see Section 3.3.4).

Edmonds and Karp [63] claimed that Dijkstra implemented as Algorithm 3.6 runs in $O(|V||A|)$ time. However, they included no proof for this claim. In reaction to the claim of Edmonds and Karp, Johnson [113] presented a family of graphs without negative cycles, for which Dijkstra actually needs $O(2^{|V|})$ iterations. Johnson stated without proof that there exists no graphs for which more iterations are needed. In that case, the number of iterations of Dijkstra’s algorithm is bounded by $\Theta(2^{|V|})$.

Before we actually describe a family of graphs for which Dijkstra’s algorithm needs an exponential number of iterations, we first prove that the number of iterations of the Generic Shortest Path algorithm (Algorithm 3.4), of which Dijkstra is a special case, is bounded by $O(2^{|V|})$. This is an extension of Theorem 3.4.

**Theorem C.6** Consider the Generic Shortest Path algorithm (Algorithm 3.4) on a digraph $G = (V, A)$ that contains no cycles of negative length. The number of iterations is $O(2^{|V|})$.

**Proof:** We extend the proof of Theorem 3.4. In our proof of that theorem, we explicitly used the property that all arc costs are nonnegative at the end of the proof only. However, it is actually not necessary that all arc costs are nonnegative to complete the proof. From the inequalities (3.2) and (3.3), it follows that

$$\sum_{n=1}^{h-1} c_{u_n, u_{n+1}} + \sum_{n=1}^{j-1} c_{w_n, w_{n+1}} < 0 \quad \text{(C.3)}$$

Recall that $u_i = w_j$ and $w_i = u_h$. Thus, the two summations in (C.3) form the negative length of the cycle $(u_i, (u_i, u_{i+1}), u_{i+1}, \ldots, (u_{h-1}, u_h), u_h = w_i, (w_i, w_{i+1}), \ldots, (w_{j-1}, w_j), w_j = u_i)$. Since we considered only graphs that contains no cycles of negative length, we found the contradiction we needed to complete the original proof.  

Instead of the graph family described by Johnson, for which Dijkstra needs an exponential number of iterations, we present another graph family described by Roos [166], which we consider to be more simple. For any $n \in \mathbb{N}_+$, a graph $G_n = (V, A)$ is defined with $n + 1$ nodes and $\frac{1}{2}n(n + 1)$ arcs as follows: $V = \{0, 1, 2, \ldots, n - 1, n\}$ and $A = \{(v, w) \in V \times V | n \geq i > j \geq 0\}$. For any arc $(v, w) \in A$, the arc cost $c_{vw} = -2^{v-1} - 2(v - w)$. The graph $G_4$ is depicted in Figure C.2.

This graph family has the nice property that one has $(v, w) \in A$ if and only if $v > w$. Thus, the graph contains no cycles and surely no cycles of negative length.
Furthermore, each node is reachable from node $n$ and for all $(v, w) \in \mathcal{A}$, we have $c_{vw} \leq -3$. Let us consider an implementation of the Generic Shortest Path algorithm (Algorithm 3.4) where, in each iteration, the pivot node $v \in \mathcal{Q}$ is selected such that $v \leq w$ for all $w \in \mathcal{Q}$. Later, we will prove that this selection rule is equivalent to Dijkstra’s selection rule for a member $G_n$ of the described graph family, if we look for the shortest path tree rooted at node $n$. The proof for the exponential behavior of Dijkstra’s algorithm, presented here, is inspired by a proof of exponential behavior of the A* algorithm by Pijls [152].

First, we will prove that the described variant of the Generic Shortest Path algorithm needs $2^n$ iterations to complete. Again, we will make use of the $\mathcal{R}$ set, defined by $\mathcal{R} = \{v \in \mathcal{V} | d_v \neq \infty \land v \notin \mathcal{Q}\}$. For ease of notation, we define the set $I(v) = \{p | p \in \mathbb{N}, v < p \leq n\}$.

**Proposition C.7** If we run the abovementioned implementation of the Generic Shortest Path algorithm on the graph $G_n$ with origin node $s = n$, we have at the end of each iteration that every finite node label $d_v$ represents the length of the path $(s = p_0, (p_0, p_1), \ldots, (p_i, v), v)$ with $p_i \in \mathcal{R} \cap I(v)$ for $0 \leq i \leq k$, and $p_i > p_j$ for all $0 \leq i < j \leq k$.

**Proof:** We prove this by induction on the iteration number $k$. After the first iteration, we have $\mathcal{R} = \{n\}$ and all labels are finite. For any $v \neq n$, we have $I(v) = \{n\}$. Thus, $\mathcal{R} \cap I(v) = \{n\}$ and $d_v$ equals the length $c_{nv}$ and represents the path $(n, (n, v), v)$. The proposition holds after the first iteration. Furthermore, all nodes have a finite label after the first iteration, thus $\mathcal{V} = \mathcal{R} \cup \mathcal{Q}$. Now, suppose that the proposition holds after iteration $k$. In the next iteration, node $v \in \mathcal{Q}$ is selected as a pivot node such that $v \leq w$ for all $w \in \mathcal{Q}$. The nodes $0, 1, \ldots, v - 1$ also have a finite label. Since they are not in $\mathcal{Q}$, they are in $\mathcal{R}$. During iteration $k + 1$, the node

![Figure C.2: The graph $G_4$, for which Dijkstra needs an exponential number of iterations to determine the shortest path tree rooted at node 4.](image)
For such a node \( w \), labels of any node \( v \) are not affected because every arc goes from a higher node number to a lower node number. So, we consider only nodes \( w \) such that \( v > w \geq 0 \).

For such a node \( w \), the label \( d_w \) at the beginning of iteration \( k+1 \) represents, by the induction hypothesis, the path \( P_{sw} = (n = p_0, (p_0, p_1), \ldots, p_h, (p_h, v - 1), (v - 1, v - 2), \ldots, w) \). We have:

\[
d_w = \sum_{(v, w) \in P_{sw}} c_{vw} = \sum_{i=0}^{h-1} c_{p_i-p_{i+1}} + c_{p_h-v} + \sum_{i=w}^{v-2} c_{i+1,i}
\]

\[
eq \sum_{i=0}^{h-1} c_{p_i-p_{i+1}} - 2^{p_h-1} - 2(p_h - (v - 1)) + \sum_{i=w}^{v-2} \{-2^i - 2(i + 1 - i)\}
\]

\[
eq \sum_{i=0}^{h-1} c_{p_i-p_{i+1}} - 2^{p_h-1} - 2p_h + 2w - \sum_{i=w}^{v-2} 2^i = \sum_{i=0}^{h-1} c_{p_i-p_{i+1}} + c_{p_h-w} - \sum_{i=w}^{v-2} 2^i
\]

and, again by the induction hypothesis:

\[
d_v + c_{vw} = \sum_{i=0}^{h-1} c_{p_i-p_{i+1}} + c_{p_h-v} + c_{vw}
\]

\[
eq \sum_{i=0}^{h-1} c_{p_i-p_{i+1}} - 2^{p_h-1} - 2(p_h - v) - 2^{v-1} - 2(v - w)
\]

\[
eq \sum_{i=0}^{h-1} c_{p_i-p_{i+1}} + c_{p_h-w} - 2^{v-1}
\]

Since \( 2^{v-1} > \sum_{i=w}^{v-2} 2^i \) for all \( 0 \leq w \leq v - 2 \), the label of node \( d_w \) will be updated during iteration \( k+1 \). Thus, node \( w \) is removed from \( \mathcal{R} \) and added to \( \mathcal{Q} \). At the end of iteration \( k+1 \), we have for each \( v > w \geq 0 \) that \( \mathcal{R} \cap \mathcal{I}(w) = \{p_0, p_1, \ldots, p_h, v\} \) and \( d_w \) represents the path \( (n = p_0, (p_0, p_1), \ldots, p_h, (p_h, v), v, (v, w), w) \). For all nodes \( v \leq w \leq n \), neither \( \mathcal{R} \cap \mathcal{I}(w) \) nor \( d_w \) and its represented path, are changed. This completes the induction proof.

**Proposition C.8** Let \( a_n-1a_{n-2}\ldots a_1a_0 \) be the binary representation of the number \( k - 1 \). This representation and the set \( \mathcal{R} \) at the end of iteration \( k \) are related by the equality: \( \mathcal{R}\{n\} = \{p\mid a_p = 1, 0 \leq p \leq n\} \).

**Proof**: We prove this relation by induction on the iteration number \( k \). After the first iteration, we have \( k - 1 = 0 \) and thus \( a_p = 0 \) for all \( 0 \leq p \leq n - 1 \). At the end of the iteration, \( \mathcal{R}\{n\} = \emptyset \). Thus, the relation holds after the first iteration. Now suppose that the relation holds at the end of iteration \( k \). During iteration \( k+1 \), the pivot node \( v \in \mathcal{Q} \) is selected with \( v \leq w \) for all \( w \in \mathcal{Q} \). By induction hypothesis \( k - 1 = a_n-1a_{n-2}\ldots a_101\ldots 1 \). As we have seen in the proof of Proposition C.7, the labels of all nodes \( w, 0 \leq w < v \) are updated. These nodes are removed from \( \mathcal{R} \) and \( v \) is added to \( \mathcal{R} \) during iteration \( k + 1 \). Adding 1 to the binary representation
of $k-1$ gives $a_{n-1}a_{n-2}\ldots a_{v+1}10\ldots 0$, which exactly describes the set $R$ at the end of iteration $k+1$. This completes the induction proof.

**Proposition C.9** The implementation of the Generic Shortest Path algorithm selecting node $v \in Q$ as a pivot node such that $v \leq w$ for all $w \in Q$, needs $2^n$ iterations to complete if $G_n$ is taken as the input graph and node $n$ is the origin.

**Proof:** The algorithm ends as soon as $Q = \emptyset$, or equivalently $R = V$. By Proposition C.8, this will be the case after iteration $k$, for which the binary representation of $k-1$ has the form $a_{n-1}a_{n-2}\ldots a_0$ and $a_i = 1$ for all $0 \leq i \leq n-1$. Thus, $k-1 = 2^n - 1$ and $k = 2^n$ iterations are needed.

**Theorem C.10** Let $G(V,A)$ be a digraph with no cycles of negative length. *Dijkstra* runs in $\Theta(2^{|V|})$ time.

**Proof:** By Theorem C.6, the Generic Shortest Path algorithm (of which *Dijkstra* is a special case), needs $O(2^{|V|})$ iterations. Thus, we only have to prove there exists a graph for which *Dijkstra*’s algorithm actually needs $O(2^{|V|})$ graphs. By Proposition C.9, the implementation of the Generic Shortest Path algorithm selecting node $v \in Q$ as a pivot node such that $v \leq w$ for all $w \in Q$, needs $2^{|V|}-1$ operations. To adapt this result for *Dijkstra*, we only have to prove that the described selection rule resembles *Dijkstra*’s selection rule for the considered graph. We will prove that, at the start of iteration $k$, we have:

$$v < w \Rightarrow d_v < d_w \forall v,w \in Q. \quad (C.4)$$

Surely, this relation holds at the start of the first iteration, since $Q = \{n\}$. Now, suppose that (C.4) holds at the start of an iteration $k \geq 1$. In iteration $k$, node $v \in Q$ is selected such that $v \leq w$ for all $w \in Q$. By the induction hypothesis, we have $d_v < d_w$ for all $w \in Q \setminus \{v\}$. As we have seen in the proof of Proposition C.7, the label of any node $w$ with $n \geq w \geq v$ is not changed during iteration $k$. On the other hand, the label of any node $w$ with $v > w \geq 0$ will be changed to $d_w + c_{vw} = d_v - 2^{v-1} - 2(v-w)$. During iteration $k$, only node $v$ is removed from $Q$, while the nodes $0, 1, \ldots, v-1$ are added to $Q$. For any pair of nodes that were already in $Q$ at the start of iteration $k$, (C.4) holds by the induction hypothesis. Since all arc costs are negative, we have $d_w < d_v$ for all updated node labels $d_w$. It can readily be seen that for any pair of nodes $u$ and $w$, such that $v > w > u \geq 0$, we have $d_u < d_w$. Hence, (C.4) holds at the start of iteration $k+1$. This completes the induction proof.

After node $n$ is selected as a pivot node, the following pivot nodes are $0, 1, 0, 2, 0, 1, 0, 3, 0, 1, 0, 2, 0, 1, 0, \ldots$. As observed by Pijls [152], this series is also known from the ‘Towers of Hanoi’ (described by Lucas [131]), a famous topic that is frequently used in computer science to illustrate recursive programming.
C.3 Floyd-Warshall

Verifying whether or not all entries of a given distance matrix $D$ represent shortest path lengths by inspecting the inequalities (C.1), takes $\Theta(|V|^3)$ time. Therefore, it is nice that there exists an implementation of Algorithm C.2 with a running time $O(|V|^3)$. This implementation is the well-known FLOYD-WARSHALL algorithm, invented by Floyd [68] and based on a theorem of Warshall [196].

We follow [35] in the description of FLOYD-WARSHALL. Let $G = (V, A)$ be a digraph with no negative cycles. The node set $V$ can be written as $V = \{v_1, v_2, \ldots, v_{|V|}\}$. We define the set $V^k \subset V$ (1 \leq k \leq |V|) as $V^k = \{v_1, v_2, \ldots, v_k\}$ and we define $V^0 = \emptyset$.

For any path $P_{vw} = (u_1 = v, (u_1, u_2), u_2, \ldots, (u_{n-1}, u_n), u_n = w)$, the intermediate nodes of $P_{vw}$ are $u_2, u_3, \ldots, u_{n-1}$.

For any pair of nodes $v, w \in V$ we consider all $v$-$w$ paths of which the intermediate nodes are all in $V^k$. Let $P^k_{vw}$ be a shortest path among those paths. There is a relation between $P^k_{vw}$ and the shortest paths with all intermediate nodes in $V^{k-1}$. This relation depends on whether or not node $v_k$ is an intermediate node of $P^k_{vw}$.

- If $v_k$ is not an intermediate node of $P^k_{vw}$, then all intermediate nodes of $P^k_{vw}$ are in $V^{k-1}$. Thus, the length of $P^k_{vw}$ equals the length of $P^{k-1}_{vw}$ if $v_k$ is not an intermediate node of $P^k_{vw}$.

- If $v_k$ is an intermediate node of $P^k_{vw}$, then $P^k_{vw} = (u_1 = v, (u_1, u_2), u_2, \ldots, u_m = v_k, \ldots, (u_{n-1}, u_n), u_n = w)$. All intermediate nodes of the subpath $P_{v,v_k} = (u_1 = v, (u_1, u_2), u_2, \ldots, (u_{m-1}, u_{m}), u_m = v_k)$ and the subpath $P_{v_k,w} = (u_m = v_k, (u_m, u_{m+1}), \ldots, (u_{n-1}, u_{n}), u_n = w)$ are in $V^{k-1}$. From Theorem 2.41, we know that both subpaths are shortest paths themselves (here, among all paths whose intermediate nodes are in $V^{k-1}$). Thus, $\ell(P^k_{vw}) = \ell(P^k_{v,v_k}) + \ell(P^k_{v_k,w})$.

The length of $P^k_{vw}$ is denoted by $d^k_{vw}$. If $k = 0$, we only consider paths consisting of one edge at most. Thus, $d^0_{vw}$ takes the form of (C.2). For $k \geq 1$ we have:

$$d^k_{vw} = \min\left(d^{k-1}_{vw}, d^{k-1}_{v,v_k} + d^{k-1}_{v_k,w}\right)$$

Clearly, $d^{|V|}_{vw}$ is the length of a shortest $v$-$w$ path in $G$, since all intermediate nodes are in $V = V$. FLOYD-WARSHALL (Algorithm C.3) iterates over $k$ to determine the values $d^k_{vw}$ for all pairs of nodes $v, w \in V$. Conveniently, one can overwrite the information of the previous iteration $k - 1$. Thus, only a memory storage of $O(|V|^2)$ is needed.

Since the ‘labels’ $d_{vw}$ will become permanent all at once at the end of the algorithm, this algorithm can be considered a label-correcting algorithm. We will mention two implementation details that might improve the running time of the original algorithm. The first improvement is rather straightforward. Between the last two “for-loops”, a check is inserted (See Algorithm C.4). If either $v = v_k$ or $d_{v,v_k} = \infty$ the last “for-loop” can be skipped.
Algorithm C.3: Floyd-Warshall ($\mathcal{G}$)
1: Initialize APSP ($\mathcal{G}, \mathbf{D}$)
2: for $k \leftarrow 1$ to $|\mathcal{V}|$ do
3:     foreach $v \in \mathcal{V}$ do
4:         foreach $w \in \mathcal{V}$ do
5:             $d_{vw} \leftarrow d_{v,v_k} + d_{v_k,w}$

Algorithm C.4: Floyd-Warshall (variant 1) ($\mathcal{G}$)
1: Initialize APSP ($\mathcal{G}, \mathbf{D}$)
2: for $k \leftarrow 1$ to $|\mathcal{V}|$ do
3:     foreach $v \in \mathcal{V}\backslash\{v_k\}$ do
4:         if $d_{v,v_k} \neq \infty$ then
5:             foreach $w \in \mathcal{V}$ do
6:                 $d_{vw} \leftarrow d_{v,v_k} + d_{v_k,w}$

In our experiments, it turns out to be profitable to determine the set of nodes $\mathcal{V}^+_{v_k}$ directly after the first “for-loop” in Algorithm C.4. The set $\mathcal{V}^+_{v_k}$ contains all nodes $v$ other than $v_k$, for which $d_{v,v_k} \neq \infty$. Using this set, the last “for-loop” will run faster if $|\mathcal{V}^+_{v_k}| < |\mathcal{V}|$. However, in dense graphs, the extra work to determine $\mathcal{V}^+_{v_k}$ (lines 3–6) (which takes $O(|\mathcal{V}|)$ time) might be higher than the profit gained in the last “for-loops”. The resulting algorithm (Algorithm C.5) still runs in $O(|\mathcal{V}|^3)$ time.

Algorithm C.5: Floyd-Warshall (variant 2) ($\mathcal{G}$)
1: Initialize APSP ($\mathcal{G}, \mathbf{D}$)
2: for $k \leftarrow 1$ to $|\mathcal{V}|$ do
3:     $\mathcal{V}^+_{v_k} \leftarrow \emptyset$
4:     foreach $v \in \mathcal{V}\backslash\{v_k\}$ do
5:         if $d_{v,v_k} < \infty$ then
6:             $\mathcal{V}^+_{v_k} \leftarrow \mathcal{V}^+_{v_k} \cup \{v\}$
7:     foreach $v \in \mathcal{V}\backslash\{v_k\}$ do
8:         if $d_{v,v_k} \neq \infty$ then
9:             foreach $w \in \mathcal{V}^+_{v_k}$ do
10:                $d_{vw} \leftarrow d_{v,v_k} + d_{v_k,w}$

C.4 Johnson

Another well-known algorithm to solve the APSP is the algorithm by Johnson [115]. Basically, this algorithm reweighs the arc costs, such that they become nonnegative.
Next, the Generic Shortest Path algorithm (Algorithm 3.4) is executed $|V|$ times, such that each node $v \in V$ is taken once as the origin node. Although Johnson described an efficient priority queue implementation for dense networks and a generalized class of the Generic Shortest Path algorithm based on arc set partitions as well (see Section 3.4), his work is mostly referred to for reweighing the arc costs.

The reweighing procedure of Johnson is based on Proposition 3.13. The idea is to determine a potential function $f$, such that $c_f(v, w) = c_{vw} + f_v - f_w \geq 0$ for all $(v, w) \in A$. Subsequently a node $s$ is temporarily added to the graph $G = (V, A)$ with outgoing arcs $(c, v)$ for all $v \in V$. These new arcs have zero arc cost. The Bellman-Ford-Moore algorithm (Algorithm 3.7)) is executed with $s$ as origin node. By Proposition C.4, it can now be detected whether or not $G$ contains a cycle of negative length. Note that the expanded graph (containing node $s$) contains no cycle that includes $s$ (since $s$ has only outgoing arcs). Thus, if a cycle of negative length is detected, the original graph $G$ contains that cycle. If a negative cycle is detected, the algorithm of Johnson is stopped.

Suppose $G$ contains no cycles of negative length. In that case, each node $v \in V$ now has a node label $d_v$. If $d$ is used as the potential function $f$, we now have $c_f(v, w) = c_{vw} + d_v - d_w$. Clearly, $d_w - d_v \leq c_{vw}$. Thus $c_f(v, w) \geq 0$ for all $(v, w) \in A$. Next, any implementation of the Generic Shortest Path algorithm can be used to determine the reduced shortest path lengths $d_f(v, w)$ between every pair of nodes $(v, w) \in A$. Especially the label-setting algorithms will now run in $O(|V|)$ iterations for each origin node, where the original negative arc costs might have lead to an exponential number of iterations (see Section C.2.2).

At the end, a postprocessing step is needed to determine the original shortest path lengths between any pair of nodes. The original length of a shortest $v$-$w$ path is $d_f(v, w) - f_v + f_w$. Clearly, this postprocessing step takes $\Theta(|V|^2)$ time.

### C.5 Preprocessing and the APSP

In this section, we will describe a preprocessing technique of the graph $G = (V, A)$. We mention two algorithms for solving the APSP that explicitly use the results of this preprocessing. These algorithms were firstly presented to solve the so-called Simple Temporal Problem (STP), which is a special case of the Temporal Constraint Satisfaction Problem (TCSP). The TCSP in turn is a special case of the Constraint Satisfaction Problem (CSP).

A survey on the CSP can be found in [190]. The TCSP and the STP are firstly presented in [44]. Dechter and Pearl [47] introduced the notion of directional path-consistency and proposed an algorithm to turn an input graph into a so-called directional path-consistent graph. To formalize the CSP, TCSP and STP, we would need a vast amount of new notation. Since we are especially interested in the resulting APSP algorithms, we rewrite the algorithms according to the notation used in this thesis. However, we will use the names of the algorithms as originally stated.
It turns out that another recent preprocessing technique, called Contraction Hierarchies (Section 6.4), very closely resembles the ideas of directional path-consistency. We briefly describe the differences in Section C.5.2. The efficiency of shortest path algorithms that make use of either Contraction Hierarchies or directional path consistency, relies strongly on the node ordering that is used. In Section 6.4.3, we already described several heuristics to determine a node ordering. In Section C.5.3, we will describe another one, called Minimum Degree Ordering and we will present a fast implementation of that heuristic.

### C.5.1 Directional path-consistency

The notion of directional path-consistency is commonly used in the literature on the CSP.

**Definition C.11** Let $G = (V, A)$ be a digraph and let an ordering '<' on the nodes be given. A graph $G = (V, A)$ is directional path-consistent if for any pair of arcs $(v_x, v_y)$ and $(v_y, v_z)$, such that $v_x > v_y$ and $v_z > v_y$, there exists an arc $(v_x, v_z) \in A$ with $c(v_x, v_z) \leq c(v_x, v_y) + c(v_y, v_z)$.

The ordering '<' on the nodes in $V$ is commonly expressed as $v_i < v_j$ if and only if $i < j$. Note that we have rewritten the original definition of directional path-consistency, because the original definition is stated in notation and definitions related to the literature about CSP, which differs from the notation and definitions we use in this thesis. From Definition C.11, it is clear that there is a strong relation between directional path-consistency and bypassing a node (see Definition 6.8). In [47], a preprocessing algorithm DPC is presented, which turns any graph $G = (V, A)$ into a directional path-consistent graph by updating arc costs and adding arcs to $G$.

**Algorithm C.6: DPC($G', <$)**

1: for $k = 1$ to $|V|$ do
2:   foreach $(v_i, v_k) \in \delta^{-}(v_k)$ do
3:     if $v_k < v_i$ then
4:       foreach $(v_k, v_j) \in \delta^{+}(v_k)$ do
5:         if $v_k < v_j$ and $v_i \neq v_j$ then
6:           if $(v_i, v_j) \in A$ then
7:             $c(v_i, v_j) \leftarrow \text{min}(c(v_i, v_j), c(v_i, v_k) + c(v_k, v_j))$
8:           else
9:             $A \leftarrow A \cup \{(v_i, v_j)\}$
10:            $c(v_i, v_j) \leftarrow c(v_i, v_k) + c(v_k, v_j)$

**Proposition C.12** Let $G = (V, A)$ be a graph and '<' be an ordering on the nodes in $V$. The graph $G'$ that results after the execution of DPC($G', <$) is directional path-consistent with respect to the node ordering '<'.
Proof: We prove this by contradiction. Suppose \( G' = (\mathcal{V}, \mathcal{A}') \) is not directional path-consistent. Thus, two arcs \((v_x, v_y)\) and \((v_y, v_z)\) exist with \(v_x > v_y\) and \(v_z > v_y\), such that either \((v_x, v_z) \notin \mathcal{A}'\) or the arc \((v_x, v_z) \in \mathcal{A}'\) has cost \(c'(v_x, v_z) > c'(v_x, v_y) + c'(v_y, v_z)\). Clearly, in iteration \(k = y\), either \((v_x, v_z)\) was added to \(\mathcal{A}'\) or the arc cost \(c'(v_x, v_z)\) was updated to \(c'(v_x, v_y) + c'(v_y, v_z)\). During any later iteration, the arc cost \(c'(v_x, v_z)\) can only be decreased. At the end of the DPC algorithm, we have \((v_x, v_z) \in \mathcal{A}'\) and \(c'(v_x, v_z) \leq c'(v_x, v_y) + c'(v_y, v_z)\). This contradicts our assumption, hence \(G'\) is directional path-consistent. \(\square\)

Proposition C.13 The length of a shortest path between nodes \(s, t\) in a graph \(G = (\mathcal{V}, \mathcal{A})\) equals the length of a shortest path between nodes \(s\) and \(t\) in \(G'\), the directional path-consistent version of \(G\) resulting from the execution of the DPC algorithm.

Proof: During the execution of the DPC algorithm, none of the arcs in \(\mathcal{A}\) are deleted. All \(s\)-\(t\) paths in \(G = (\mathcal{V}, \mathcal{A})\) still exist in \(G' = (\mathcal{V}, \mathcal{A}')\). However, new arcs might have been added to \(\mathcal{A}'\) or the arc costs of any arc in \(\mathcal{A}\) might have been decreased. Any new arc \((v_i, v_j)\) or any update of the arc costs \((v_i, v_j)\) is based on a path \(P = (v_i, v_k, v_k, (v_k, v_j), v_j)\), which already exists in \(G'\) at that moment. The arc cost of arc \((v_i, v_j)\) is set to \(\ell(P)\) and thus \(c'(v_i, v_j) \geq d(v_i, v_j)\). Hence, no shorter paths are introduced.

As stated in line 5 of Algorithm C.6, a shortcut \((v_i, v_j)\) that represents a loop \((v_i = v_j)\) is not added to \(\mathcal{A}\). In the literature about CSP, such shortcuts are inspected to check whether or not \(c(v_i, v_k) + c(v_k, v_i) < 0\). A negative loop shows the existence of a cycle of negative length. Although other algorithms that solve the Negative Cycle Detection Problem (see Section C.2.1) are usually faster, while running DPC it can easily be checked as well whether or not \(G\) contains a cycle of negative length.

Proposition C.14 Let \(G = (\mathcal{V}, \mathcal{A})\) be a digraph and ‘\(<\)’ an ordering on the nodes. Let \(G_1\) be the upward graph and \(G_1\) be the downward graph based on the directional path-consistent graph \(G'\) that results after the execution of the DPC(G, <) algorithm. Let \(P_{v_i,v_i}\) be a shortest path in \(G\). There exists a path \(P' = (v_k = u_0, u_0, u_1, \ldots, u_p, \ldots, (u_{k-1}, u_k), u_k = v_1)\) in \(G'\) that has the same length as \(P\) where \((u_i, u_{i+1}) \in \mathcal{A}_1\) for \(0 \leq i \leq p - 1\) and \((u_i, u_{i+1}) \in \mathcal{A}_1\) for \(p \leq i \leq k - 1\).

Proof: From Proposition C.13, we know that a \(v_s\)-\(v_t\) path exists in \(G'\) with length \(\ell(P)\). We therefore only have to prove that a \(v_s\)-\(v_t\) path of the form \(P'\) exists. Let \(P' = (v_s = u_0, (u_0, u_1), u_1, \ldots, (u_{k-1}, u_k), u_k = v_1)\) be a shortest path in \(G'\) that does not have the form of \(P'\). Let \(u_p (1 < p < k)\) be the first node in \(P'\), such that \(u_p-1 > u_p\) and \(u_{p+1} > u_p\). Such a node \(u_p\) exists, since \(P'\) is not of the form of \(P'\). Since \(G'\) is directional path-consistent and \(P'\) is a shortest path, we have \((u_{p-1}, u_{p+1}) \in \mathcal{A}'\) and \(c'((u_{p-1}, u_{p+1}) = c'(u_{p-1}, u_p) + c'(u_p, u_{p+1})\). Let \(\bar{P}\) be the path that results if the subpath \(((u_{p-1}, u_p), (u_p, u_{p+1}))\) of \(P'\) is replaced by the arc \((u_{p-1}, u_{p+1})\). \(\bar{P}\) lies in \(G'\) and has the same length as \(P'\), but contains one node
Chleq [31] presented an algorithm Min-Path (Algorithm C.7) for the SSSD-SPP that takes the forward and backward graph of a directional path-consistent graph (and implicitly its related ordering ‘<’) as input.

Algorithm C.7: Min-Path(\(G_u, G_d, <, v_s, v_t\))

1: foreach \(v \in \mathcal{V}\) do  
2: \(d_v \leftarrow \infty\)
3: \(d_{v_s} \leftarrow 0\)
4: for \(k = s\) to \(|\mathcal{V}|\) do
5: \(d_{v_j} \leftarrow \min(d_{v_j}, d_{v_k} + c(v_k, v_j))\)
6: for \(k = |\mathcal{V}|\) down to \(t\) do
7: \(d_{v_j} \leftarrow \min(d_{v_j}, d_{v_k} + c(v_k, v_j))\)

As in Yen’s improvement on Bellman-Ford-Moore (Section 6.4), we first iterate in the forward graph over \(v_s, v_{s+1}, \ldots, v_{|\mathcal{V}|}\) (line 4) and then in the backward graph over \(v_{|\mathcal{V}|}, v_{|\mathcal{V}|-1}, \ldots, v_t\) (line 7). There are two main differences compared to Yen’s improvements. The main difference is that in Bellman-Ford-Moore, there are still \(|\mathcal{V}| - 1\) passes needed. After pass \(n\), the shortest path from node \(v_s\) to any other node that contains at most \(n\) arcs is determined. Furthermore, Chleq’s algorithm does not iterate over all nodes but starts in the forward graph from \(v_s\). In the backward graph, it terminates at \(v_t\). Clearly, Chleq’s algorithm exploits the directional path-consistency of the graph \(\mathcal{G}\).

Theorem C.15 At the end of the execution of the Min-Path algorithm, \(d_{v_s} = d(v_s, v_t)\).

Proof: We will prove the result by induction. Our induction hypothesis is that, at the start of iteration \(k\) in the third main “for-loop” (line 7), we have \(d_{v_n} = d(v_s, v_n)\) for all \(k \leq n \leq |\mathcal{V}|\). At the start of the first iteration (\(k = |\mathcal{V}|\)), the second “for-loop” (line 4) has just been finished. From Proposition C.14, we know there exists a shortest path \(P = (v_s = u_0, (u_0, u_1), u_1, \ldots, u_{n-1}, u_n = v_{|\mathcal{V}|})\) in the directional path-consistent graph \(\mathcal{G}\), such that \(u_i < u_j\) for all \(0 \leq i < j \leq |\mathcal{V}|\). This, or an equivalent path, will surely be determined during the first “for-loop”. Since for every \(u_i \in P\), the arc \((u_i, u_{i+1})\) is scanned. Either the node label \(d_{u_{i+1}}\) is updated or the node label \(d_{u_i}\) equals \(d_{u_i} + c(u_i, u_{i+1})\) and is based on another \(v_s-v_{i+1}\) path with the same length. At the start of the first iteration, we therefore have \(d_{v_{|\mathcal{V}|}} = d(v_s, v_{|\mathcal{V}|})\) and the induction hypothesis holds for iteration \(k = |\mathcal{V}|\).

Now, suppose that the induction hypothesis holds at the start of iteration \(k\) (\(k > 1\)) in the third main “for-loop”. We will prove that, at the start of iteration \(k - 1\), we...
have \( d_{v_{k-1}} = d(v_s, v_{k-1}) \). From Proposition C.14, we know that a shortest \( v_s-v_{k-1} \) path exists of the form \( (v_s = u_0, \ldots, u_p, \ldots, u_{n-1}, u_n, v_{k-1}) \) such that \( (u_i, u_{i+1}) \in A_i \) for \( 0 \leq i < p \), and \( (u_i, u_{i+1}) \in A_i \) for \( p \leq i < n \). If \( p = n \), the label \( d_{v_{k-1}} \) already equals \( d(v_s, v_{k-1}) \) after the second “for-loop” finished. If \( p < n \), we have by the induction hypothesis that \( d_{u_i} = d(v_s, u_i) \) for all \( p \leq i < n \). After iteration \( k \), the arc \( (u_{n-1}, u_n) \in A \) is surely scanned. Hence, \( d_{v_{k-1}} = d(v_s, v_{k-1}) \).

This completes the induction proof.

The shortest path distance from node \( v_s \) to any other node \( v \in V \) can be determined by executing Algorithm C.7 once, taking \( v_t = v_1 \). At the end of the algorithm, \( d_{v_i} = d(v_s, v_i) \) for all \( 1 \leq i \leq |V| \). Thus, Algorithm C.7 can be executed \( |V| \) times to get a solution for the APSP.

In Chleq’s algorithm, it might happen during the second main “for-loop” that \( d_{v_k} = \infty \) (line 4). In that case, it does not make sense to execute the sub “for-loop” (line 5). This observation results in a variant of Chleq’s algorithm (Algorithm C.8).

**Algorithm C.8: Chleq (Variant 1) \((G_\uparrow, G_\downarrow, <, v_s, v_t)\)**

1: foreach \( v \in V \) do
2: \( d_v \leftarrow \infty \)
3: \( d_{v_s} \leftarrow 0 \)
4: for \( k = s \) to \( |V| \) do
5: if \( d_{v_k} < \infty \) then
6: foreach \( (v_k, v_j) \in A_\uparrow \) do
7: \( d_{v_j} \leftarrow \min(d_{v_j}, d_{v_k} + c(v_k, v_j)) \)
8: for \( k = |V| \) down to \( t \) do
9: foreach \( (v_k, v_j) \in A_\downarrow \) do
10: \( d_{v_j} \leftarrow \min(d_{v_j}, d_{v_k} + c(v_k, v_j)) \)

The second main “for-loop” can also be implemented using a priority queue \( Q \), as in Dijkstra’s algorithm. A recent algorithm by Delling et al. [48] is in fact a variant of Chleq’s Min-Path algorithm, using such a priority queue. Delling et al. used Contraction Hierarchies in order to get a “weakly directional path-consistent” graph (as we will see in Section C.5.2, Theorem C.15 also holds for weakly directional path-consistent graphs). Delling et al. proposed a sophisticated parallel implementation of their algorithm, called PHAST, an acronym of ‘parallel hardware-accelerated shortest path trees’. PHAST can be executed on a graphics processing unit (GPU). GPU’s are known to be very powerful and highly-parallel and are nowadays also used for general-purpose computations [145].

These shortest path algorithms (Algorithms C.7, C.8 and PHAST) can easily be adapted to situations where we are interested in the shortest path distances from a node \( s \) to a set of destinations \( T \subset V \). Delling et al. [51] proposed such an adaption. They updated their PHAST algorithm to RPHAST (restricted PHAST) for this type of calculation. Their main idea is that, instead of the downward digraph \( G_\downarrow \),
its induced subgraph $G_2'(T', A')$ is used. The set $T'$ is determined as follows: The process is initialized by $T' \leftarrow T$. Based on the reversed graph $G_2$, a full execution of the Generic Shortest Path algorithm is started. The candidate set $Q$ is initialized with $T$. Hence, every node $v \in T$ starts with $d_v = 0$. If during the search, a label of a node $w \notin T'$ is updated, node $w$ is added to $T'$. Clearly, $T \subseteq T' \subseteq V$. Determining $T'$ this way ensures that the shortest paths from node $s$ to all nodes in $T$ are found if the induced subgraph $G'_2(T', A')$ is used. Note, that in case we are interested in the shortest paths between two given sets of origin nodes $S$ and destination nodes $T$, the algorithm is executed $|S|$ times, but the set $T'$ has to be determined only once.

Planken et al. [156] use DPC as an initialization step in their Snowball algorithm (Algorithm C.9). Where DPC bypasses nodes one-by-one, based on a certain ordering ‘$<$’ of the nodes, Snowball uses the reverse ordering and iterates over $v_1, v_2, \ldots, v_{|V|}$. During the iteration of $v_i$, the shortest path distances from $v_i$ to $v_j$ and from $v_j$ to $v_i$ are determined for all $i < j \leq |V|$.

Algorithm C.9: Snowball($G, <$)

1: Initialize APSP ($G, D$)
2: for $k = |V|$ down to 1 do
3:  foreach $(v_k, v_i) \in A_1$ do
4:     foreach $j \in \{k + 1, k + 2, \ldots, |V|\}$ do
5:         $d_{v_k, v_j} \leftarrow \min(d_{v_k, v_j}, c_{v_k, v_i} + d_{v_i, v_j})$
6:  foreach $(v_i, v_k) \in A_1$ do
7:     foreach $j \in \{k + 1, k + 2, \ldots, |V|\}$ do
8:         $d_{v_i, v_k} \leftarrow \min(d_{v_i, v_k}, d_{v_j, v_k} + c_{v_j, v_k})$

Theorem C.16 If $G$ is directional path-consistent with respect to the ordering ‘$<$’, then Algorithm C.9 correctly determines the shortest path length between all pairs of nodes, provided that $G$ contains no cycles of negative length.

Proof: We will prove by induction that, at the end of iteration $k$ (at line 8), $d_{v_i, v_j} = d(v_i, v_j)$ for all $1 \leq i \leq |V|$ and $k \leq j \leq |V|$. At the end of the first iteration (when $k = |V|$), we have $d_{v_1, v_{|V|}} = 0$. Since $G$ contains no cycles of negative length, the condition holds at the end of the first iteration. Now, assume that the condition holds after any iteration $k > 1$. For any $k \leq j \leq |V|$, it follows from Proposition C.14 that a shortest $v_{k-1} - v_j$ path exists that is the concatenation of one arc $(v_{k-1}, v_i)$ and a shortest $v_i - v_j$ path $P$ that contains only nodes $v_n$, such that $v_n \geq v_i$. From the induction hypothesis, we have $d_{v_i, v_j} = \ell(P)$. For each $j$ ($k \leq j \leq |V|$), all possible concatenations are considered (lines 3-5). Thus, at the end of iteration $k - 1$ we have $d_{v_{k-1}, v_j} = d(v_{k-1}, v_j)$. Similarly, for any $k \leq j \leq |V|$ there exists a shortest $v_j - v_{k-1}$ path that is the concatenation of a shortest $v_j - v_i$ path (which contains only nodes $v_n \geq v_j$) and an arc $(v_j, v_k) \in A_1$. All possible concatenations are considered in lines 6-8. Hence, at the end of iteration $k - 1$, we
have \(d_{v_i,v_{j-1}} = d(v_j,v_{k-1})\) for \(k \leq j \leq k\). Combined with the induction hypothesis and the initialized value \(d_{v_{k-1},v_{k-1}} = 0\), it follows that, at the end of iteration \(k - 1\), we have \(d_{v_i,v_j} = d(v_i,v_j)\) for all \(k - 1 \leq i \leq |V|\) and \(k - 1 \leq j \leq |V|\). This completes the proof.

\[\square\]

**Proposition C.17** Algorithm C.9 runs in \(O(|V||A|)\) time.

**Proof:** All arcs in \(A\) are visited exactly once during the execution of the main “for-loop” (line 2) and the “for-loop” in line 3. The “for-loop” in line 4 takes \(O(|V|)\) time. Hence, the total work in the upward direction takes \(O(|V||A\uparrow|)\) time. Similarly, the work in the downward direction takes \(O(|V||A\downarrow|)\) time. Since \((A\uparrow,A\downarrow)\) is a partitioning of \(A\), the algorithm runs in \(O(|V||A|)\) time.

\[\square\]

It should be noted that Proposition C.17 explicitly states that the input graph needs to be directional path-consistent (in order to ensure the correctness of the Snowball algorithm). An arbitrary graph \(G\) (containing no cycles of negative length) can be turned into a directional path consistent graph with respect to an ordering ‘\(<\’ by executing Algorithm C.6. During that execution, several shortcuts might be added to the original graph. Hence, the running time of \(O(|V||A|)\) mentioned in the proposition, is not stated in terms of the original graph \(G\). However, for some graph families like chordal graphs\(^1\), there is an ordering for which the directional path consistent graph \(G\)' of the original graph contains the same arc set as the input graph. Determining whether or not a graph is chordal and determining the ordering such that no new arcs are added, can both be done in \(O(|V| + |A|)\) time [188] by using the Maximum Cardinality Search algorithm (See Algorithm C.10). This algorithm maintains buckets \(B_i (i = 0, 1, \ldots, |V| - 1)\). In \(B_i\), all nodes are stored with exactly \(i\) adjacent nodes that have not yet been chosen as a pivot node. At the end of the Maximum Cardinality Search algorithm, the vector \(\pi\) determines the ordering ‘\(<\’: if \(\pi_v < \pi_w\) then \(v < w\). Tarjan and Yannakakis proved that, by using this ordering, there are no new arcs added while running DPC, if the graph is chordal. For chordal graphs, Snowball (that solves the APSP) has the same complexity as the Bellman-Ford-Moore algorithm (that determines only one shortest path tree).

It turned out to be profitable to add a simple check whether or not an arc is redundant. The idea is based on the arc reduction technique, described in Section 6.3.1.2. However, the implementation here is more straightforward. In Algorithm C.11, the redundancy check is added (in lines 4 and 8). The check in line 4 skips arcs \((v_k,v_i)\) for which there exists an arc \((v_k,v_j) \in A\uparrow\), such that \(c_{v_k,v_j} + d(v_j,v_i) < c_{v_k,v_i}\). Note that \(d(v_j,v_i)\) represents the length of a shortest \(v_j-v_i\) path. Clearly, none of the shortest paths from \(v_k\) to any node \(v (v > v_k)\) will contain \((v_k,v_i)\). Hence, this arc can be omitted. A similar argument holds for the check in line 8.

Proposition C.17 shows the importance of finding a node ordering ‘\(<\’ that adds only a small number of shortcuts to the original graph, in order to make the graph

---

\(^1\)Chordal graphs arise in the study of Gaussian elimination [167].
Algorithm C.10: **Maximum Cardinality Search**\((G = (V, E))\)

1: for \(i = 0\) to \(|V| - 1\) do  
2: \(B_i \leftarrow \emptyset\)  
3: foreach \(v \in V\) do  
4: \(B_i \leftarrow B_i \cup \{v\}\)  
5: \(d_i \leftarrow 0\)  
6: \(b \leftarrow 0\)  
7: for \(v = |V|\) down to 0 do  
8: Select a node \(v\) from \(B_b\)  
9: \(B_b \leftarrow B_b - \{v\}\)  
10: \(\pi_v \leftarrow q\)  
11: \(d_v \leftarrow -1\)  
12: foreach \(\{v, w\} \in E\) do  
13: if \(d_w \neq -1\) then  
14: \(B_{d_w} \leftarrow B_{d_w} - \{w\}\)  
15: \(d_w \leftarrow d_w + 1\)  
16: \(B_{d_w} \leftarrow B_{d_w} \cup \{w\}\)  
17: while \(b < |V| - 1\) and \(B_b = \emptyset\) do  
18: \(b \leftarrow b + 1\)

Algorithm C.11: **Snowball (Arc Reduction)**\((G, <)\)

1: **Initialize** APSP \((G, D)\)  
2: for \(k = |V|\) down to 1 do  
3: foreach \((v_k, v_i) \in A^+\) do  
4: if \(c_{v_k,v_i} \leq c_{v_k,v_j} + d(v_j, v_i)\) \(\forall (v_j, v_i) \in A^+\) then  
5: foreach \(j \in \{k+1, k+2, \ldots, |V|\}\) do  
6: \(d_{v_k,v_j} \leftarrow \min(d_{v_k,v_j}, c_{v_k,v_i} + d_{v_i,v_j})\)  
7: foreach \((v_i, v_k) \in A^-\) do  
8: if \(c_{v_i, v_k} \leq d(v_i, v_j) + c_{v_j, v_k}\) \(\forall (v_j, v_k) \in A^-\) then  
9: foreach \(j \in \{k+1, k+2, \ldots, |V|\}\) do  
10: \(d_{v_j,v_k} \leftarrow \min(d_{v_j,v_k}, d_{v_j,v_i} + c_{v_i,v_k})\)
directional path-consistent. Several heuristics to determine such a node ordering are described in Section C.5.3.

C.5.2 Contraction Hierarchies and DPC

A recent preprocessing technique, called Contraction Hierarchies (Section 6.4), turns out to be closely related to directional path-consistency. Algorithm 6.7 describes the Contraction Hierarchies preprocessing. The only difference with DPC (Algorithm C.6) is the additional check in line 6 of Algorithm 6.7.

Based on this additional check, the addition of a shortcut or the update of an arc cost, is omitted if a witness path (Definition 6.10) exists. In that case, the condition of Definition C.11 is violated and hence, the resulting graph is not directional path-consistent. However, it turns out that if the resulting graph is only weakly directional path-consistent, the results described in this paragraph are still valid.

Definition C.18 Let $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ be a digraph and let an ordering ‘$<$’ on the nodes be given. A graph $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ is weakly directional path-consistent if for any pair of arcs $(v_x, v_y)$ and $(v_y, v_z)$, such that $v_x > v_y$ and $v_z > v_y$, there exists a $v_x$-$v_y$ path $P$ containing only nodes $v > v_y$ with $\ell(P) \leq c(v_x, v_y) + c(v_y, v_z)$.

Clearly, any directional path-consistent graph is weakly directional path-consistent as well. Proposition C.14 also holds for weakly directional path-consistent graphs:

Proposition C.19 Let $\mathcal{G} = (\mathcal{V}, \mathcal{A})$ be a digraph and ‘$<$’ an ordering on the nodes. Let $\mathcal{G}_\uparrow$ be the upward graph and $\mathcal{G}_\downarrow$ be the downward graph based on the weakly directional path-consistent graph $\mathcal{G}'$ resulting from the execution of the Contraction Hierarchies($\mathcal{G}, <$) algorithm. Let $P_{v_i, v_j}$ be a shortest path in $\mathcal{G}$. There exists a path $P' = (v_0, (u_0, u_1), \ldots, v_y, \ldots, (u_{k-1}, u_k), u_k = v_t)$ in $\mathcal{G}'$ with the same length as $P$ where $(u_i, u_{i+1}) \in \mathcal{A}_\uparrow$ for $0 \leq i \leq p - 1$ and $(u_i, u_{i+1}) \in \mathcal{A}_\downarrow$ for $p \leq i \leq k - 1$.

Proof: This follows directly from Proposition 6.12.

The similarity between Proposition C.19 and Proposition 6.12 shows the relation between (weak) directional path-consistency and Contraction Hierarchies. The SSSD-SPP algorithm, which is based on Contraction Hierarchies (Algorithm 6.8) can be seen as a bidirectional variant of the Min-Path algorithm (Algorithm C.7).

C.5.3 Minimum-Degree Ordering

As stated before, the ordering ‘$<$’ on the nodes strongly influences the efficiency of the resulting algorithms. One goal is to minimize the number of extra arcs needed to ensure that the resulting graph is (weakly) directional path-consistent. Yannakakis [202] proved that determining a node ordering, which results in the minimum number
of extra arcs, is NP-hard. Several heuristics are proposed in order to find a good node ordering. A well-known heuristic is the so-called Minimum-Degree Ordering, proposed by Rose [168]. This algorithm (See Algorithm C.12) takes an undirected graph $G = (V, E)$ as input. In case of a directed graph, we can use the underlying undirected graph as input. At the end of the algorithm, the ordering $<$ is described as follows: $v < w$ if and only if $\pi_v < \pi_w$. 

**Algorithm C.12: Minimum-Degree Ordering ($G = (V, E)$)**

1: $n \leftarrow |V|$  
2: for $k \leftarrow 1$ to $n$ do 
3: Let $v \in V$ be a node with minimum degree  
4: $\pi_v \leftarrow i$  
5: foreach $\{u,v\} \in E$ do 
6: foreach $\{v,w\} \in E$ do 
7: if $u \neq w$ and $\{u,w\} \notin E$ then 
8: $E \leftarrow E \cup \{u,w\}$  
9: $E \leftarrow E - \{u,v\}$  
10: $V \leftarrow V - \{v\}$

The idea is that nodes are selected in order of the minimum degree. As soon as node $v$ is selected, the graph $G$ is updated by removing node $v$ and its incident edges. Furthermore, some shortcuts $\{u,w\}$ might be added to $E$. Both $u$ and $w$ were adjacent to node $v$ just before the graph was updated and $\{u,w\} \notin E$. The performance of this algorithm strongly depends on both the data structure used for determining the node with minimum degree, and on the data structure that implements the graph $G$. We propose an efficient data structure for these purposes. We use two $|V| \times |V|$ matrices to maintain the graph $G$: $A$ and $B$. Furthermore, we store the node degree of each node in a vector $d$. This data structure can be initialized in $O(|V|^2 + |E|)$ time by running Algorithm C.13. We assume that there exists a mapping $f: V \rightarrow [0, 1, \ldots, |V| - 1]$ and we just write $d[v]$ to denote $d[f(v)]$ and $A[v, w]$ to denote $A[f(v), f(w)]$.

At the end of Algorithm C.13, clearly $d[v]$ equals the degree of node $v$. For each node $v$, the adjacent nodes are stored in the positions $A[v, 0], A[v, 1], \ldots, A[v, d[v] - 1]$. If $A[v, i] = w$ (for $0 \leq i < d[v]$), then $B[v, w] = i$. Now suppose a node $v$ has to be extracted from $G$. Recall that node $v$ and its incident edges are removed from $G$, while some shortcuts might have to be added to $G$ in order to preserve the possible paths via node $v$ in $G$. Algorithm C.14 describes these steps more formally. The idea is that only the degree of nodes adjacent to node $v$ is influenced. Let $w$ be a node adjacent to $v$. First, node $w$ loses an adjacent node (node $v$). The corresponding updates are described in lines 3–6. Next, node $w$ can obtain new neighbors (due to the addition of shortcuts to $E$). The corresponding updates are described in lines 9–13. Line 10 checks whether or not the addition of the shortcut $\{u, w\}$ is needed. Clearly, these update steps take $O(d[v]^2)$ time.
Algorithm C.13: Initialize data structures \( (G = (V, E)) \)

1: foreach \( v \in V \) do
2: \( d[v] \leftarrow 0 \)
3: foreach \( w \in V \) do
4: \( B[v, w] \leftarrow -1 \)
5: foreach \( \{v, w\} \in E \) do
6: \( A[v, d[v]] \leftarrow w \)
7: \( B[v, w] \leftarrow d[v] \)
8: \( d[v] \leftarrow d[v] + 1 \)
9: \( A[w, d[w]] \leftarrow v \)
10: \( B[w, v] \leftarrow d[w] \)
11: \( d[w] \leftarrow d[w] + 1 \)

Algorithm C.14: Update data structures \( (G = (V, E), v) \)

1: for \( i \leftarrow 0 \) to \( d[v] - 1 \) do
2: \( w \leftarrow A[v, i] \)
3: \( d[w] \leftarrow d[w] + 1 \)
4: \( u \leftarrow A[w, d[w]] \)
5: \( B[w, u] \leftarrow B[w, v] \)
6: \( A[w, B[w, u]] \leftarrow u \)
7: for \( j \leftarrow 0 \) to \( d[v] - 1 \) do
8: if \( i \neq j \) then
9: \( u \leftarrow A[v, j] \)
10: if \( B[w, u] = -1 \) then
11: \( B[w, u] \leftarrow d[w] \)
12: \( A[w, d[w]] \leftarrow u \)
13: \( d[w] \leftarrow d[w] + 1 \)
The data structure we use for determining the node with minimum degree that has to be selected, is a non-monotonic priority queue (see Definition 6.14). In our implementation, each possible node degree 0, 1, ..., |V|−1 is represented by a bucket. Furthermore, we use a pointer n to the first non-empty bucket. This idea closely resembles Dial’s bucket data structure (see Appendix B.2.1). Clearly, the operations Increase-Key and Decrease-Key can be implemented in \(O(1)\) time. Suppose node \(v\) is selected in Algorithm C.12. A neighbor \(w\) of node \(v\) \((d[w] \geq d[v], \text{before node } v \text{ is removed})\) might lose at most one neighbor (that is node \(v\)). Hence, after updating the graph, the pointer \(n\) might be decreased by at most one. Let \(\alpha\) be the number of times that \(n\) takes a different value. Note that \(n\) might have to be redetermined in each iteration of Algorithm C.12. The range of \(n\) is clearly \([0, 1, \ldots, |V|−1]\). Furthermore, node \(n\) can be decreased by at most one during each iteration. Hence, \(\alpha\) is bounded by \(2|V|\). Thus, the Delete-Min operation runs in \(O(1)\) amortized time.

George and Liu [83] mention several enhancements on the Minimum-Degree Ordering algorithm. They also show that the Minimum-Degree Ordering is surprisingly sensitive to the initial ordering of the nodes.

In Section 6.4.3, we mentioned several node-ordering heuristics that are used with Contraction Hierarchies. Of course, these heuristics can be used for the APSP as well. However, for the APSP that is only executed once for a graph, finding a good node-ordering focuses mainly on the minimal number of arcs added to maintain directional-path consistency. In Contraction Hierarchies one large-scale graph is used as input for many SSSD-SPP’s. As said before, these SSSD-SPP’s paths will be solved faster if the node ordering shows a uniform numbering over the graph. Furthermore, since many SSSD shortest path problems will be solved on the graph, the preprocessing of the graph might take some extra time.

## C.6 Experimental evaluation

Planken et al. [156] include an extensive experimental evaluation to show the effectiveness of their Snowball algorithm. They made their test cases publicly available. In all experiments, they compared the following algorithms:

- **Floyd-Warshall**
- **Johnson** (Fibonacci-heap)
- **Chleq**
- **Snowball**

Here, Chleq is the algorithm that runs the Min Path algorithm \(|V|\) times. Each time a different source node \(v_s\) is selected. The destination node \(v_d\) is always set to \(v_1\). This way, Chleq solves the APSP.

We use their test cases to compare several other implementations as well. The test cases considered are mentioned in Table C.1.
Table C.1: Test instances from Planken et al. [156]

| Code | Type                        | #Cases | |V|   | |A|                        |
|------|-----------------------------|--------|-----------------|------------------|-----------------|
| T1   | Chordal - constant nodes    | 250    | 1,000           | 151,680–998,980  |
| T2   | Chordal - variable nodes    | 130    | 214–3,125       | 45,576–1,279,018 |
| T3   | Scale-Free - constant nodes| 130    | 1,000           | 3,992–134,720    |
| T4   | Scale-Free - variable nodes| 160    | 250–1,000       | 4,352–6,660      |
| T5   | New-York                    | 170    | 108–3,906       | 226–12,844       |
| T6   | Diamonds                    | 130    | 111–2,751       | 222–5,502        |
| T7   | Job-shop                    | 400    | 17–1,321        | 64–220,440       |
| T8   | Hierarchical Task Networks  | 121    | 500-625         | 1,496–3,198      |

The following algorithms are used in our evaluation: FLOYD-WARSHALL and its two variants (Section C.3), several implementations of JOHNSON (Section C.4), and algorithms based on preprocessing of the graph (Section C.5). The latter consists of CHILEQ as described above together with its variant (that is based on Algorithm C.8 instead of Algorithm C.7) and SNOWBALL (Algorithm C.9) and its variant based on arc reduction (Algorithm C.11). Johnson’s algorithm is implemented with several heaps: d-heap (Section B.1.1), binomial heap (Section B.1.2), Fibonacci heap (Section B.1.3) and a relaxed heap (Section B.1.4). We implemented the d-heap in two ways. One version only considers d-heaps with \(d = 2^i\) for \(n \geq 1\). This way, bitwise operations can be used to further increase the calculation speed (especially for determining the position of the first child and the parent of a node). Furthermore, we combined Johnson’s algorithm with several label-correcting algorithms (Sections 3.3.1 and 3.3.2): BELLMAN-FORD-MOORE, D’ESOPO-PAPE, PALLOTTINO and SMALL LABEL FIRST. Some of these are combined with optimizations (Section 3.3.3) like parent-checking (P) and Large Label Last (LLL). This way, we tested 32 different implementations for the APSP.

Although the arc costs in the test sets are integral, we did not implement any bucket-based data structure here. The value \(C\) (the highest arc cost) is always (much) larger than \(|V|\). Thus, we might not expect any improvement of the running time using a bucket-based priority queue.

In our experiments, we use the MINIMUM-DEGREE ORDERING heuristic of Rose [167] (Algorithm C.12) to determine the node ordering. Only for the test set ‘Chordal’, we used the MAX CARDINALITY SEARCH procedure (Algorithm C.10). Planken et al. used the same ordering procedures, thus their results should be comparable.

Table C.2 shows the cumulated run times for each test set (T1, T2, ..., T8). Before we discuss the test sets in more detail, we first highlight some results from this table. In each column, the shortest run time for the corresponding test set is underlined. For four test sets, either SNOWBALL or SNOWBALL (ARC REDUCTION) is the fastest. For three other test sets, a label-setting variant of JOHNSON is the fastest. Note that this is always a d-heap implementation where so-called bitwise operations are used (hence, \(d\) is a power of 2). Only for one test set, a label-correcting variant of JOHNSON runs fastest.
Table C.2: Cumulated run times (seconds) for all test sets

<table>
<thead>
<tr>
<th>Method</th>
<th>T1</th>
<th>T2</th>
<th>T3</th>
<th>T4</th>
<th>T5</th>
<th>T6</th>
<th>T7</th>
<th>T8</th>
</tr>
</thead>
<tbody>
<tr>
<td>DPC-based</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Chleq</td>
<td>1,457.1</td>
<td>1,100.4</td>
<td>712.4</td>
<td>54.9</td>
<td>71.5</td>
<td>15.9</td>
<td>378.3</td>
<td>13.0</td>
</tr>
<tr>
<td>Chleq (V)</td>
<td>1,284.0</td>
<td>793.2</td>
<td>694.6</td>
<td>49.6</td>
<td>47.6</td>
<td>12.5</td>
<td>297.9</td>
<td>11.2</td>
</tr>
<tr>
<td>Snowball</td>
<td>1,430.4</td>
<td>1,025.1</td>
<td>533.2</td>
<td>21.5</td>
<td>33.9</td>
<td>9.9</td>
<td>268.1</td>
<td>4.4</td>
</tr>
<tr>
<td>Snowball (AR)</td>
<td>1,326.7</td>
<td>647.8</td>
<td>710.0</td>
<td>25.6</td>
<td>33.8</td>
<td>9.9</td>
<td>257.0</td>
<td>4.9</td>
</tr>
<tr>
<td>Johnson-LS</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2^1-heap</td>
<td>1,073.2</td>
<td>952.5</td>
<td>93.1</td>
<td>30.6</td>
<td>109.6</td>
<td>27.7</td>
<td>127.4</td>
<td>13.2</td>
</tr>
<tr>
<td>2^2-heap</td>
<td>1,062.7</td>
<td>933.9</td>
<td>89.2</td>
<td>28.6</td>
<td>99.4</td>
<td>27.3</td>
<td>123.2</td>
<td>12.1</td>
</tr>
<tr>
<td>2^3-heap</td>
<td>1,062.2</td>
<td>931.6</td>
<td>91.2</td>
<td>30.4</td>
<td>108.6</td>
<td>27.2</td>
<td>124.1</td>
<td>13.1</td>
</tr>
<tr>
<td>2-heap</td>
<td>1,085.2</td>
<td>961.3</td>
<td>96.2</td>
<td>31.9</td>
<td>111.4</td>
<td>28.6</td>
<td>130.9</td>
<td>13.7</td>
</tr>
<tr>
<td>3-heap</td>
<td>1,096.5</td>
<td>967.8</td>
<td>98.4</td>
<td>32.5</td>
<td>109.6</td>
<td>29.7</td>
<td>133.1</td>
<td>13.5</td>
</tr>
<tr>
<td>4-heap</td>
<td>1,067.7</td>
<td>939.2</td>
<td>91.3</td>
<td>29.7</td>
<td>101.7</td>
<td>27.9</td>
<td>125.1</td>
<td>12.3</td>
</tr>
<tr>
<td>5-heap</td>
<td>1,086.8</td>
<td>960.4</td>
<td>97.7</td>
<td>32.7</td>
<td>115.6</td>
<td>29.2</td>
<td>131.5</td>
<td>13.9</td>
</tr>
<tr>
<td>6-heap</td>
<td>1,083.1</td>
<td>958.0</td>
<td>97.4</td>
<td>32.6</td>
<td>115.2</td>
<td>29.2</td>
<td>131.0</td>
<td>13.9</td>
</tr>
<tr>
<td>7-heap</td>
<td>1,080.0</td>
<td>957.0</td>
<td>97.6</td>
<td>32.5</td>
<td>114.1</td>
<td>29.2</td>
<td>130.5</td>
<td>13.9</td>
</tr>
<tr>
<td>8-heap</td>
<td>1,062.9</td>
<td>939.6</td>
<td>92.3</td>
<td>30.7</td>
<td>109.5</td>
<td>27.9</td>
<td>125.7</td>
<td>13.3</td>
</tr>
<tr>
<td>Binomial h.</td>
<td>1,110.4</td>
<td>993.9</td>
<td>112.6</td>
<td>41.5</td>
<td>155.7</td>
<td>32.5</td>
<td>144.3</td>
<td>18.5</td>
</tr>
<tr>
<td>Fibonacci h.</td>
<td>1,147.9</td>
<td>1,032.3</td>
<td>124.3</td>
<td>46.2</td>
<td>177.1</td>
<td>36.7</td>
<td>155.7</td>
<td>19.9</td>
</tr>
<tr>
<td>Relaxed h.</td>
<td>1,260.8</td>
<td>1,173.5</td>
<td>167.7</td>
<td>58.7</td>
<td>210.3</td>
<td>43.6</td>
<td>185.5</td>
<td>24.8</td>
</tr>
<tr>
<td>Johnson-LC</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFM</td>
<td>3,275.1</td>
<td>2,860.5</td>
<td>150.1</td>
<td>28.0</td>
<td>210.9</td>
<td>24.9</td>
<td>374.6</td>
<td>10.8</td>
</tr>
<tr>
<td>BFM (P)</td>
<td>2,842.6</td>
<td>2,349.5</td>
<td>137.7</td>
<td>27.0</td>
<td>142.2</td>
<td>25.6</td>
<td>261.7</td>
<td>10.4</td>
</tr>
<tr>
<td>BFM (LLL)</td>
<td>2,263.0</td>
<td>2,096.6</td>
<td>127.1</td>
<td>26.8</td>
<td>82.5</td>
<td>30.6</td>
<td>252.5</td>
<td>9.6</td>
</tr>
<tr>
<td>BFM (LLL, P)</td>
<td>2,145.2</td>
<td>1,907.7</td>
<td>123.4</td>
<td>26.4</td>
<td>84.1</td>
<td>32.3</td>
<td>214.6</td>
<td>9.8</td>
</tr>
<tr>
<td>EP</td>
<td>5,531.5</td>
<td>5,172.1</td>
<td>158.2</td>
<td>23.4</td>
<td>94.3</td>
<td>25.2</td>
<td>344.2</td>
<td>8.7</td>
</tr>
<tr>
<td>EP (LLL)</td>
<td>3,449.4</td>
<td>3,266.6</td>
<td>135.5</td>
<td>21.6</td>
<td>84.4</td>
<td>32.2</td>
<td>255.7</td>
<td>8.8</td>
</tr>
<tr>
<td>Pall.</td>
<td>5,257.1</td>
<td>4,971.9</td>
<td>161.2</td>
<td>24.4</td>
<td>77.6</td>
<td>25.3</td>
<td>344.1</td>
<td>9.1</td>
</tr>
<tr>
<td>Pall. (LLL)</td>
<td>3,403.5</td>
<td>3,227.6</td>
<td>137.7</td>
<td>22.3</td>
<td>80.3</td>
<td>32.1</td>
<td>258.1</td>
<td>9.0</td>
</tr>
<tr>
<td>SLF</td>
<td>2,670.4</td>
<td>2,193.1</td>
<td>113.2</td>
<td>21.4</td>
<td>109.3</td>
<td>24.6</td>
<td>288.3</td>
<td>8.9</td>
</tr>
<tr>
<td>SLF (P)</td>
<td>2,546.2</td>
<td>2,065.5</td>
<td>112.9</td>
<td>22.3</td>
<td>103.2</td>
<td>26.4</td>
<td>251.1</td>
<td>9.0</td>
</tr>
<tr>
<td>SLF (LLL)</td>
<td>2,053.1</td>
<td>1,782.5</td>
<td>105.0</td>
<td>22.3</td>
<td>77.2</td>
<td>28.4</td>
<td>213.1</td>
<td>8.9</td>
</tr>
<tr>
<td>SLF (LLL, P)</td>
<td>2,030.3</td>
<td>1,742.0</td>
<td>105.4</td>
<td>22.9</td>
<td>80.5</td>
<td>30.1</td>
<td>203.7</td>
<td>9.1</td>
</tr>
<tr>
<td>Floyd-W.</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>FW (original)</td>
<td>1,903.7</td>
<td>4,828.3</td>
<td>985.6</td>
<td>410.1</td>
<td>9,317.4</td>
<td>2,874.8</td>
<td>705.6</td>
<td>161.8</td>
</tr>
<tr>
<td>FW (V1)</td>
<td>1,905.1</td>
<td>4,858.8</td>
<td>961.5</td>
<td>389.7</td>
<td>5,012.8</td>
<td>505.8</td>
<td>687.6</td>
<td>127.4</td>
</tr>
<tr>
<td>FW (V2)</td>
<td>2,273.4</td>
<td>5,807.5</td>
<td>1,129.0</td>
<td>450.0</td>
<td>4,258.9</td>
<td>38.4</td>
<td>793.1</td>
<td>132.3</td>
</tr>
</tbody>
</table>
Comparing the three implementations of Floyd-Warshall shows that the second variant is only effective on the relatively sparse instances that are included in the test sets T5 (New York) and T6 (Diamonds). Especially on test set T6 (Diamonds), the reduction of the original running time is huge (98.7%), making this implementation competitive to some label-setting variants of Johnson’s algorithm. However, for not so sparse graph instances the running time is higher than the original version. Clearly, variant one of Floyd-Warshall’s algorithm can be seen as an ‘in-between’ variant. For very sparse graphs, variant 1 is outperformed by variant 2. However, for some less sparse graph instances variant 1 can outperform the original version.

Johnson’s algorithm, implemented as a label-setting algorithm, runs, the fastest if a $d$-heap is used where $d$ is a power of 2, such that fast bitwise operations can be used in the implementation. In these test sets, the 4-heap that is implemented this way always outperforms the binary heap. Although the Fibonacci heap and the Relaxed heap have the best bounds on the running time (See Appendix B), they are outperformed by all other heap implementations, including the Binomial heap, which in turn is outperformed by the $d$-heap. As mentioned in Appendix B, in practice, the Fibonacci heap is often outperformed by other, theoretically less efficient, data structures [23, 35]. The same seems to hold for the Relaxed heap.

For chordal graphs (test sets T1 and T2), the label-correcting variants of Johnson’s algorithm cannot compete with its label-setting variants. Surprisingly, the D’Esopo-Pape and Pallottino implementations are outperformed by Bellman-Ford-Moore on these test sets. For the Bellman-Ford-Moore variants on all test sets, the Parent-Check and the Large Label Last optimizations perform better than the original version. Combining these two optimizations results in the fastest Bellman-Ford-Moore variants, except for test set T8 (Hierarchical Task Networks).

Chleq algorithm is always outperformed by Snowball. However, our proposed variant of Chleq is always an improvement to the original version and for the scale free test sets (T1 and T2) this variant even outperforms Snowball. Our proposed variant of Snowball, which uses arc reduction, reduces the running time of four test sets. On the other four test sets, the running time is somewhat increased.

In the next subsections, we will discuss the test sets in more detail and compare our results with the ones found by Planken et al. [156].

### C.6.1 Chordal graphs (test sets T1 and T2)

There are two test sets (T1 and T2) that contain chordal graphs. Test set T1 contains 250 graphs with $|V| = 1000$. However, the number of arcs varies, based on the notion of tree width$^2$. For each different number of arcs, there are 10 graphs included in test set T1. The number of arcs varies from 151,680 up to 998,980.

In Figure C.3, the cumulated running times are shown for each different number of

---

$^2$A formal definition of tree width can be found in [165].
The all-pairs shortest path problem

0.2 0.4 0.6 0.8 1

timing in milliseconds (log-scale)

Floyd-Warshall
Chleq (Variant)
Snowball (AR)
Johnson (2²-h.)

Figure C.3: Results for the test set T1, where all chordal graphs have $|V| = 1,000$

There are 25 different numbers of arcs and for each number of arcs there are 10 graphs included in the test set. The graph shows the results for Floyd-Warshall, Chleq (Variant), Snowball (Arc Reduction) and Johnson with a 4-heap that makes explicit use of bitwise operations.

Our evaluation on this test set shows another result than the one found by Planken et al. In our experiments, there is an intersection between all pairs of lines in Figure C.3. Planken et al. found only intersections between Floyd-Warshall and every other line. They observed that Snowball outperformed Chleq and Chleq in turn outperformed Johnson on all instances. One difference between their evaluation and ours is that they used the Fibonacci-heap for Johnson’s algorithm. But this does not explain the intersection between Chleq and Snowball that we found.

Furthermore, based on Table C.2, even Johnson’s algorithm with a Fibonacci heap outperforms Chleq and Snowball in our evaluation.

Test set T2 contains chordal graphs $G = (V, E)$ for which $|\delta(v)| \leq 211$ for all $v \in V$. However, the cardinality of $|V|$ is varied. There are 13 different values for $|V|$ among the graphs in the test set. For each value, there are 10 graphs included in the test set. In Figure C.4, the results are shown in more detail for Floyd-Warshall, Chleq (Variant), Snowball (Arc Reduction) and Johnson with a 4-heap (implemented with bitwise operations).

These results are comparable to those obtained by Planken et al. However, Snowball (here, with arc-reduction) outperforms Johnson by a much smaller factor than in their experiments.
C.6 Experimental evaluation

10.4 16.6 10.8 14.2 10.2 16.6 12.4 11.2 10.4

Figure C.4: Results for the chordal graphs in test set T2

C.6.2 Scale-free networks (test sets T3 and T4)

A scale-free network contains many nodes that have only a few adjacent nodes and a few nodes that have many adjacent nodes. More formally:

Definition C.20 A scale-free network is a graph of which its degree distribution follows a power law, at least asymptotically. That is, the fraction \( P(k) \) of nodes in the graph with \( k \) adjacent nodes grows for large values of \( k \) as: \( P(k) \sim ck^{-\gamma} \) for some constant \( c \), called the normalization constant, and a parameter \( \gamma \) (typically, \( 2 < \gamma < 3 \)).

Several real-world graphs, such as in social-networks, biological networks and World Wide Web links are (or were) considered to be scale-free [8]. We explicitly write ‘were’, since these considerations are still discussed [200].

Two test sets contain scale-free networks. In test set T3, all instances have \( |V| = 1,000 \) and a varying number of arcs. For each number of arcs, there are 10 graphs included in the test set. Figure C.5 shows the cumulative running times for different cardinalities of \( |A| \). This figure closely resembles the results of Planken et al., except that Johnson’s algorithm outperforms Chleq (here implemented with our variant) and Snowball earlier (hence, for smaller \(|A|\)).

Test set T4, which also contains scale-free networks, consists of graphs with a varying number of nodes. Figure C.6 shows the results for \textsc{Floyd-Warshall}, \textsc{Chleq (Variant)}, \textsc{Snowball} and \textsc{Johnson} (using \textsc{Small Label First}).
Figure C.5: Results for the scale-free network test set with $|V| = 1,000$

Figure C.6: Results for the scale-free network test varying number of nodes
The results show a decrease in the run time for both Snowball and Chleq (Variant) around $|V| = 800$. This is recognized by Planken et al. as an artifact of the benchmark generator. Planken et al. observed that Johnson (with a Fibonacci-heap) only outperforms Chleq and Snowball for small values of $|V|$. From Figure C.6, it is clear that in our results Johnson outperforms Chleq (Variant) on all instances. Furthermore, Snowball only outperforms Johnson for large values of $|V|$.

C.6.3 Test set T5: ‘New-York’

The test set ‘New-York’ (T5) is based on the road map graph of New-York, which can be obtained from the DIMACS challenge website\(^3\). This road map graph contains 264,346 nodes and 733,846 arcs. Planken et al. derived several subgraphs from the road map graph by running a breadth-first search from a random node $s$, until a desired number of nodes had been visited. 170 Subgraphs were obtained this way. For each number of nodes, there are 10 graphs included in the test set.

In Figure C.7, the running times of four algorithms are shown in more detail. For each different $|V|$, the cumulated running time is shown for the problem instances with that number of nodes. The four algorithms considered are the fastest versions of Floyd-Warshall (Variant 2), Chleq (Variant 1), Snowball (arc Reduction) and Johnson (with Small Label First and Large Label Last).

\(^3\)http://www.dis.uniroma1.it/~challenge9/
Figure C.7 closely resembles the results found by Planken et al. However, the gap between Johnson (the way we implemented it here) and Snowball (here with arc reduction) becomes less obvious for larger $|\mathcal{V}|$.

C.6.4 Test set T6: ‘Diamonds’

Test set T6 contains very sparse graphs that might occur in so-called Simple Temporal Problems. The graph takes the form of a circular chain of ‘diamonds’. Each diamond consists of two parallel paths of equal length, starting from a node and ending in another node. From the latter node, two paths start again to converge on a third node. This pattern is repeated for each diamond in the chain. The final node is then connected to the first one. The sizes of each diamond and the total number of diamonds are varied among the graphs in the test set. Figure C.8 shows the results in more detail for four implementations.

As observed before, variant 2 of Floyd-Warshall performs very good on sparse graphs. For graphs as sparse as in this test set, Snowball is the fastest algorithm, followed by Chleq. This is also observed by Planken et al.

C.6.5 Test set T7: ‘Jobshop’

Test set T7 consists of 400 graphs that are based on jobshop scheduling problems. The number of nodes varies from 17–1321. The number of arcs varies from 32–
C.6 Experimental evaluation

Figure C.9: Results for the Jobshop test set (T7)

For each number of nodes, 10 graphs are included in the test set. Planken et al. observed in their experiments that Floyd-Warshall runs the fastest for instances with $|V| < 160$ and that the difference between Johnsons, Chleq and Snowball is not quite as pronounced, although they found Snowball to be consistently the fastest of the tree by a small margin.

Figure C.9 shows our results in more detail for four implementations. Only for very small instances, Floyd-Warshall is the fastest among these four implementations. Indeed, the performance of the three other algorithms is quite similar. Yet in our experiments, Johnson (implemented with a 4-heap, that takes explicitly advantage of so-called bitwise operations), outperforms Chleq and Snowball for the larger problem instances.

C.6.6 Test set T8: ‘Hierarchical Task Networks’

Test set T8 consists of 121 graphs that imitate so-called sibling-restricted Simple Temporal Networks, which originates from Hierarchical Task Networks. In these graphs, constraints (arcs) occur only between parent tasks and their children, and between sibling tasks. The number of nodes varies from 500–625 and the number of arcs from 748–1599. For each different number of nodes, 11 graphs are included in the test set.

Planken et al. observed in their experiments that Johnson comes close to Chleq the larger instances. However, both algorithms were clearly outperformed by Snow-
ball. Figure C.10 shows our results for four algorithms in more detail. Based on Table C.2, we decided to highlight a label-correcting version of JOHNSON (based on D’ESOPO-PAPE). Clearly, this version outperforms CHLEQ algorithm for the larger instances and comes close to SNOWBALL. However, SNOWBALL is clearly the fastest algorithm for all graph instances in this test set. Although it is questionable whether or not this will be the case for larger graph instances.
D

Time Dependent Shortest Path Problem

In Section 6.5, we encountered situations where a set of cost functions $C$ is used. Each cost function $C \in C$ assigns a constant cost to each arc $(v, w) \in A$ of the graph $G = (V, A)$. Depending on the day type, time of the day, and maybe other factors like the weather forecast, a cost function $C \in C$ is chosen that resembles the actual traffic situation the best. Until now, during each specific shortest path calculation all arc costs were considered to be constant with respect to time.

Suppose we have a set of cost functions that forecast the arc costs for each 15 minute interval during the day. If it is believed (but yet unknown) that a certain trip that will start at 10:00 am takes about two hours, one might use the average of all 8 arc functions corresponding to the 15 minute intervals for the time window from 10:00 am up to 12:00 pm. Clearly, this approach has some drawbacks:

- Since the time duration is unknown, it is unclear which arc cost functions should be taken into account in order to determine an average cost function.
- Especially if the time period is long, it might happen that the average cost is not so accurate. For example if the time period starts during rush hours, but ends after the rush hours, the deviation of the arc costs considered during the time period might be high.

Modern car navigation systems recalculate the trip every few seconds. This can be recognized by remarks of the system like 'due to the current traffic, your route has been changed'. Before the initial calculation and before the recalculation, the arc cost function that describes the current traffic the best is chosen.

In vehicle routing problems, often a time table between several origins and destinations is needed. If we need a time table in a real-time environment, we recalculate this table every few seconds using the actual arc costs as input. In Section 6.5 a
hybrid preprocessing is described that can be used in such a real-time environment. Sometimes we need to solve a vehicle routing problem with data in the future. In case we have a set of arc costs for different time intervals, we can use the average cost function as described above.

There exists some shortest path algorithms that take time dependent arc costs into account in a more sophisticated way. In this appendix, we briefly describe some literature on this problem. In Section D.1 the so-called ‘Time Dependent Shortest Path Problem’ (TDSPP) is introduced as well as some theory on its complexity. The TDSPP can be combined with $A^*$, as shown in Section D.2. Even bidirectional search can be used with the TDSPP (Section D.3). Finally, in Section D.4 it is explained how Contraction Hierarchies can be used with the TDSPP.

D.1 TDSPP, theory and complexity

The TDSPP was first introduced by Cooke and Halsey [34] in 1966. They motivated their research by stating that most of the research on the SPP so far, fall seriously short of reality by assuming the arc cost to be constant. They stated that this assumption is certainly not true in many physical and biological applications. We can easily extend their examples with real road networks, where the fastest path has to be determined.

Consider a strongly connected digraph $G = (V, A)$. Let $c_{vw}(t)$ be the travel time (cost) on arc $(v, w) \in A$ leaving node $v$ at time $t$. In this section $t$ denotes a certain point in time. Here, $z$ denotes the destination node (instead of $t$). In [34] it is assumed that all $c_{vw}(t) \in \mathbb{Z}_+$ and $T = \{t_0, t_0 + 1, t_0 + 2, \ldots\}$ is an index set that contains a discrete time scale. The question is to find the shortest (here: fastest) $s$-$z$ path that starts at $t_0$.

Note that, one might also assume that all $c_{vw}(t)$ are multiples of some positive unit $\Delta$ and $T = \{t_0, t_0 + \Delta, t_0 + 2\Delta, \ldots\}$. For ease of notation we consider $\Delta = 1$.

For each node $v \in V$, the optimal value function $f_v : T \rightarrow \mathbb{Z}_+$ is defined, such that $f_v(t)$ is the minimum trip time over all $v$-$z$ paths leaving node $v$ at time $t \in T$. Cooke and Halsey modified Bellman’s Equation system (2.20) as follows:

$$\begin{align*}
    f_z(t) &= 0 & \forall t \in T \\
    f_v(t) &= \min_{(w,u) \in A} \left\{ c_{vw}(t) + f_w(t + c_{vw}(t)) \right\} & \forall v \neq z, \forall t \in T
\end{align*}$$

(D.1)

The problem is to find $f_s(t_0)$. Where (2.20) is stated in a forward way, the modified version (D.1) is stated in a backward way (taking $f_z(t) = 0$ for all $t \in T$).

Kaufman and Smith [119] observed that (D.1) implicitly defines a so-called expanded static graph $G' = (V', A')$ with $V' = \{(v,t) : v \in V, t \in T\}$ and $A' = \{((v,t),(w,u)) : (v,w) \in A; t \leq u; c_{vw}(t) = u - t\}$. Cooke and Halsey make the following necessary assumptions: $T$ is finite, and $t_0 + f_s(t_0) \leq t_0 + T_{\text{max}}$ for a known $T_{\text{max}}$. The arc
costs $c_{vw}(t)$ are modified as follows:

$$
\tilde{c}_{vw}(t) = \begin{cases} 
  c_{vw}(t), & \text{if } t + c_{vw}(t) \leq t_0 + T_{\text{max}} \\
  \infty, & \text{if } t + c_{vw}(t) > t_0 + T_{\text{max}}
\end{cases}
$$

(D.2)

In this way, arcs can only be used at times that the arrival at the end of the arc is before or at time $t_0 + T_{\text{max}}$. Cooke and Halsey solve the TDSPP by iterating over $k$:

$$
\begin{align*}
  f^k_z(t) &= 0 \\
  f^k_v(t) &= \min\left\{ f^{k-1}_v(t), \min_{(v,w) \in A} \{ \tilde{c}_{vw}(t) + f^{k-1}_w(t + \tilde{c}_{vw}(t)) \} \right\} \quad \forall \ v \neq z
\end{align*}
$$

(D.3)

Clearly this approach resembles (3.6), the original approach of Ford [70] for solving the SPP. Cooke and Halsey show that $f^k_v(t_0) = f_v(t_0)$ in a finite number of iterations, since $t_0 + f_v(t_0) \leq t_0 + T_{\text{max}}$ and all $c_{vw}(t)$ are nonnegative. Their method requires to store $\tilde{c}_{vw}(t)$ and $f_v(t)$ for all $(v, w) \in A, v \in V$ and $t \in [t_0, t_0 + 1, \ldots, t_0 + T_{\text{max}}]$, a total of $(|A| + |V|)(T_{\text{max}} + 1)$ numbers. Kaufman and Smith [119] state that the method of Cooke and Halsey runs in $O(|V|^3 T_{\text{max}})$ time. However, this is only true if there exists a simple shortest path. In that case the number of iterations is indeed bounded by $|V|$ and each iteration takes $O(|A| T_{\text{max}}) = O(|V|^2 T_{\text{max}})$ time.

Consider for example the graph in Figure D.1, taken from [144]. In this example, only arc (3,4) has a time dependent cost $c_{34}(t)$. The unique shortest path from node 1 to node 4 that starts at $t_0 = 0$ is $\{1, (1,3), 3, (3,2), 2, (2,3), 3, (3,4), 4\}$. Furthermore, the subpath $\{1, (1,3), 3, (3,2), 2\}$ is not the shortest path from node 1 to node 2 that starts at $t_0 = 0$. Hence, in the TDSPP, Theorem 2.4.1 does not hold in general.

![Figure D.1: Example where the shortest path is not simple nor the concatenation of shortest paths](image)

Based on the expanded static graph $G'$, the method of Cooke and Halsey runs in $O(|V||A| T_{\text{max}}^2)$ time.
D.1.1 The FIFO principle

Halperin [102], mentioned that Dijkstra explicitly uses the fact that nothing can be gained by a delay in the arrival time to any given node. Graphs that have this property are nowadays said to have the FIFO property.

**Definition D.1** Let $G = (V, A)$ be a digraph and $c_{vw}(t)$ be time dependent arc costs. The graph has the FIFO property if, for any $(v, w) \in A$, $t + c_{vw}(t) \leq u + c_{vw}(u)$ for all $t, u \in T$ such that $t \leq u$.

In [119] the FIFO-property is called the consistency assumption, and credited to Prakash (1989). Although the FIFO-principle sounds realistic, there are real-world situations where the FIFO-property is violated. Kaufman and Smith [119] give the following example: Traffic lights might be coordinated in such a way that traffic moving in a given direction without turning will hit a string of green lights. Suppose one car is waiting at the first traffic light of such a string, and a second car reaches the same intersection just after the traffic light becomes green. The second car will not have to stop, and will pass the first car easily, reaching the end of the arc earlier even though the second car reached the beginning of the arc later. In this example the arc considered, must have multiple lanes that allow overtaking. Orda and Rom [144] mention two other practical examples where the FIFO-property does not apply.

Dreyfus [61] modified Dijkstra such that it can be used for solving the TDSPP, see Algorithm D.1. The most important modification is that the label $d_v$ now equals the earliest possible arrival time at node $v$ if node $s$ is left at time $t_0$. Furthermore, instead of using $c_{vw}$ now $c_{vw}(d_v)$ is used in lines 10 and 11. Assuming that the FIFO-property holds and that $Q$ is implemented as a suitable priority queue (see Appendix B), the running time of Algorithm D.1 is $O(|V| \log |V| + |A|)$. Dreyfus modification cannot be used if the FIFO-property is not fulfilled. However, for a TDSPPP having the FIFO-property, a polynomial time algorithm is found. It does not matter at all whether $c_{vw}(t)$ is based on discrete or continuous values of $t$.

Orda and Rom [144] provide an example (see Figure D.2) where $c_{vw}(t)$ is defined for $t \in \mathbb{R}$ to show that in case the FIFO-property does not hold, there exist infinite shortest paths with finite path length. In their example, both $c_{12}(t)$ and $c_{21}(t)$ are non-continuous functions in $t$. If node 1 is left at time $t_0$ ($0 \leq t_0 < 1$), node 1 can be revisited a number of times: $t_k = 1 - (1 - t_0)/4^k, k = 1, 2, \ldots$. Hence, the arrival time at node 3 is $t_k + c_{13}(t_k) = 2 + (1 - t_0)/4^k > 2$. However, for $k \to \infty$, one can arrive at node 3 at time 2. Orda and Rom claim that solving the TDSPPP without the FIFO-property is NP-hard.

The FIFO-property therefore plays an important role. In [119] a check is provided to determine whether or not the FIFO-property holds. If $t$ takes discrete values, it is checked whether or not the following inequalities hold:

$$t + c_{vw}(t) \leq (t + 1) + c_{vw}(t + 1), \text{ for all } t \in T \text{ and } (v, w) \in A. \quad (D.4)$$
Algorithm D.1: Dijkstra for TDSPP $(G, s, t_0)$

1: foreach $v \in V$ do
2:     $d_v \leftarrow \infty$
3:     $p_v \leftarrow \text{NIL}$
4:     $d_s \leftarrow t_0$
5:     $Q \leftarrow \{s\}$
6: repeat
7:     Select a pivot node $v \leftarrow \arg \min \{d_v | v \in Q\}$
8:     $Q \leftarrow Q - \{v\}$
9:     foreach $(v, w) \in \delta^+(v)$ do
10:        if $d_v + c_{vw}(d_v) < d_w$ then
11:           $d_w \leftarrow d_v + c_{vw}(d_v)$
12:           $\pi_w \leftarrow v$
13:        if $w \notin Q$ then
14:           $Q \leftarrow Q \cup \{w\}$
15: until $Q = \emptyset$

Figure D.2: Example where the shortest path from node 1 to node 2, leaving node 1 at time $t_0$ ($0 \leq t_0 < 1$) contains infinite arcs.
If the FIFO-property holds for any two consecutive entries in $T$, it recursively holds for any two entries in $T$. In case $t \in \mathbb{R}$, it is assumed that $c_{vw}(t)$ is differentiable with respect to time. It is verified that the derivative always satisfies $c_{vw}'(t) \geq -1$.

Kaufman and Smith observed that the FIFO-property can easily be violated if the arc costs relate travel times to (expected) traffic volumes. If the travel time is very sensitive to the traffic volume, a large decrease in volume from one period to the next may decrease the travel time rapidly enough to violate the FIFO-property. Since one might expect that in congested traffic the order of vehicles will not change that much, one could consider to alter the arc cost function in order to satisfy the FIFO-property. For example, if $t$ takes discrete values one could update the arc costs as follows:

$$
\tilde{c}_{vw}(t+1) \leftarrow \max\{c_{vw}(t+1), c_{vw}(t) + 1\}
$$

(D.5)

### D.1.2 Speed profiles

Very recently we obtained ‘Speed Profile’ data from TomTom\footnote{http://www.tomtom.com/en_gb/licensing/products/traffic/historical-traffic/speed-profiles/} for Germany, The Netherlands and Belgium. According to TomTom this data is derived by aggregating and processing over 9 trillion anonymous GPS measurements from millions of devices (that reflect actual consumer driving patterns across the globe).

Each speed profile divides a day in 288 pieces of 5 minutes. For each piece a promillage is provided. In Figure D.3 two different speed profiles are visualized. Our data consists of 201 different speed profiles. The most important arcs (about 30%) in the road network have a so-called ‘free-flow’ speed and are linked to 7 speed profiles, one for each day of the week (possibly identical for several days). The promillage that is provided by the speed profiles should be combined with the ‘free-flow’ speed in order to forecast the actual speed at a specific day and time.

It is common that speed profiles are piecewise linear. Horn [108] introduced the linear-speed assumption. At any time $t$ the travel speed on arc $(v,w) \in A$, is the same on each position on that arc. Let $l_{vw}$ be the length of $(v,w)$, $s_{vw}(t)$ be the speed on $(v,w)$ at time $t$ and $a_{vw}(t)$ be the acceleration on $(v,w)$ at time $t$. Note that $a_{vw}(t)$ is piecewise constant, if the speed profile is piecewise linear.

Constant acceleration is subject to a well-known law of motion:

$$
\frac{1}{2}a_{vw}(t)\Delta_t^2 + s_{vw}(t)\Delta_t - \Delta_t = 0
$$

(D.6)

Note that in (D.6) both $a_{vw}(t)$ and $s_{vw}(t)$ are constants, where $t$ is the time that arc $(v,w)$ is entered. $\Delta_t$ denotes a time-period during which $a_{vw}(t)$ stays constant and $\Delta_t \leq l_{vw}$ denotes the travelled distance along $(v,w)$.

The time needed to travel a distance of $\Delta_t$ along $(v,w)$ is thus:
D.2 TDSPP and A*

If the TDSPP has the FIFO property, it can be solved by the Generic Shortest Path algorithm 3.4. It therefore makes sense to determine, whether or not enhancements

\[
\Delta t = -s_{vw}(t) + \sqrt{s_{vw}(t)^2 + 2a_{vw}(t)\Delta t}.
\]  

(D.7)

Under the linear-speed assumption, one can determine the travel time \(c_{vw}(t)\) piece-by-piece, as stated in Algorithm D.2. That algorithm follows the speed profile piece-wise. If the distance \(\Delta\ell\) that can be travelled during the remaining period of the current piece exceeds the untraveled part \(\ell\) of \(\ell_{vw}\), the time needed to travel \(\ell\) is determined in line 8 by (D.7). Otherwise, \(c_{vw}(t)\) is increased with the remaining period in the current piece, while \(\ell\) is decreased by \(\Delta\ell\) and the process continues with the next piece of the speed profile.

If for each arc \((v, w) \in A\) a piecewise linear speed profile is provided, the arc costs \(c_{vw}(t)\) determined by Algorithm D.2 imply that the FIFO-property holds. However, it is not possible anymore to determine \(c_{vw}(t)\) in \(O(1)\) time. Hence, the known bounds for solving the (static) SPP have to be multiplied by the maximum number of pieces in any speed profile if applied to the TDSPP with arc costs determined by Algorithm D.2.

D.2 TDSPP and A*
**Algorithm D.2: Piecewise determination of \(c_{vw}(t)\)**

1: \(c_{vw}(t) \leftarrow 0\)
2: \(\ell \leftarrow \ell_{vw}\)
3: \(t' \leftarrow t\)
4: while \(\ell > 0\) do
5: \(u \leftarrow \text{arg min } \{t'' > t' | a_{vw}(t') \neq a_{vw}(t'')\}\)
6: \(\Delta \ell \leftarrow s_{vw}(t')(u - t') + \frac{1}{2}a_{vw}(t')(u - t')^2\)
7: if \(\Delta \ell \geq \ell\) then
8: \(c_{vw}(t) \leftarrow c_{vw}(t) + \Delta t\)
9: else
10: \(c_{vw}(t) \leftarrow c_{vw}(t) + u - t'\)
11: \(t' \leftarrow u\)
12: \(\ell \leftarrow \ell - \ell'\)

for the Generic Shortest Path algorithm can be applied to the TDSPP as well. Horn [108] and Chabini and Lan [26] for instance, use A* (see Chapter 5) to solve the TDSPP. Horn used the traditional estimator \(h\): divide the Euclidean distance by the maximum allowed speed during the time period that is considered (see (5.7)). Chabini and Lan construct an underlying static graph:

**Definition D.2** Let \(G = (V, A)\) be a digraph and \(c_{vw}(t)\) be the time needed to traverse \((v, w) \in A\) if \(v\) is left at time \(t\). The underlying static graph \(G = (V, A)\) takes \(c_{vw}(t) = \min_t c_{vw}(t)\) as arc costs.

On this underlying static graph an all-to-one shortest path is calculated, such that \(d_v\) denotes the cost of a shortest \(v\)-\(z\) path. After this preprocessing step, the \(d_v\) values can be used as estimators during any shortest \(s\)-\(z\) calculation in the original TDSPP. Clearly, this estimates are lower bounds on \(d_v(t)\) for any time \(t\). Furthermore, these estimates are consistent (see Definition 5.5), since \(d_v - d_w\) is a lower bound on \(d(v, w)\) in the underlying static graph and \(d(v, w)\) is in turn a lower bounds on the distance between nodes \(v\) and \(w\) in the TDSSP for any possible time \(t\), we have 

\[d_t(v, w) \leq d_v - d_w.\]  
(Here, \(d_t(v, w)\) denotes the cost (travel time) of a shortest \(v\)-\(w\) path if \(v\) is left at time \(t\)).

In fact the preprocessing of Chabini and Lan can be seen as choosing a landmark (see Section 6.3.3) at node \(z\) in the underlying static graph. Combining landmarks with the TDSPP is done by Delling and Wagner [53]. Delling and Wagner choose a set of landmarks based on the underlying static graph. Although it is possible to update the stored distances from and to each landmark each time arc costs are changed, it is not necessary to do so, since the underlying static graph is based on so-called free flow speeds. Time dependency naturally arises from rush hours, accidents and traffic jams that only increase the arc costs. Hence, the landmark estimates that are based on the underlying static graph remain lower bounds in these cases.
D.3 Bidirectional methods for the TDSPP

Even if the TDSPP has the FIFO property, it looks problematic to use a bidirectional approach, since the arrival time at \( z \) is necessary to know in advance in order to run the backward search. Nannicini et al. [138, 139] present a three-phase bidirectional algorithm for the TDSPP that overcomes the unknown arrival time at \( z \). We use similar notation as in Chapter 4.

1. During the first phase bidirectional \( A^* \) (with a landmark estimator) is executed, where the forward search is run on the graph weighted by \( c_{vw}(t) \), and the backward search is based on the reversed underlying static graph. Phase 1 terminates as soon as a node \( u \) is labeled from both the forward and the backward search. An \( s-z \) path \( P \) that contains node \( u \) is found. The time dependent cost \( \mu \) of \( P \) is an upper bound on the shortest \( s-z \) path that starts at time \( t_0 \).

2. During the second phase both searches are allowed to proceed until the backward search candidate set \( Q_z \) contains only nodes with a label higher than \( \mu \). Let \( R_z = \{ v \in V_z | d_z^* \neq \infty \land v \notin Q_z \} \) contain all nodes settled during the backward search.

3. During the third phase, only the forward search is executed, with the additional constraint that only nodes in \( R_z \) are scanned.

Let \( G^*(V^*, A^*) \) be the original (forward) graph with time dependent arc costs \( c_{vw}(t) \) and let \( G^z = (V^z, A^z) \) be the reversed underlying static graph. The three phase algorithm is formally stated in Algorithm D.3.

**Theorem D.3** [139] Algorithm D.3 correctly determines the shortest \( s-z \) path in the TDSPP for a given departure time \( t_0 \).

**Proof:** Algorithm D.3 can in fact be seen as a unidirectional \( A^* \) algorithm with respect to the forward search. The only difference is that as soon as the third phase is started, some nodes are omitted (nodes that are not in \( R^z \)). Hence, we only have to prove that such an omitted node cannot be part of a shorter path than the one that is determined at the end of the algorithm. We prove this by contradiction. Suppose that Algorithm D.3 is not correct. Let \( P \) be a shortest \( s-z \) path \( P \) that starts at time \( t_0 \) with cost \( c(P, t_0) \). Let \( v^* \) be the first node on this path that is not explored during the forward search (of all three phases) of Algorithm D.3. From phase 3, we have \( v^* \notin R^z \), hence \( v^* \) is not settled from the backward search during phase 1 and 2. Let \( \beta \) be the smallest label in \( Q^z \) at the end of phase 2. Clearly, \( \beta \leq d_{z^*} + h^z_{v^*} \) and \( c(P, t_0) \leq \mu < \beta \leq d_{z^*} + h^z_{v^*} \leq d^z(s, v^*) + d^z(v^*, z) = c(P, t_0) \), which is a contradiction. Hence, Algorithm D.3 is correct. \( \square \)
Algorithm D.3: Bidir. A* for TDSPP($G^s, G^z, s, z, h^s, h^z$)

1: $u \leftarrow \text{NIL}$
2: INITIALIZE-SINGLE-SOURCE($G^s, s, Q^s$)
3: INITIALIZE-SINGLE-SOURCE($G^z, z, Q^z$)
4: ITERATE*($G^s, Q^s, h^s$)
5: ITERATE*($G^z, Q^z, h^z$)

// Phase 1
6: \textbf{repeat}
7: \hspace{1em} if a certain selection rule holds then
8: \hspace{2em} ITERATE*($G^s, Q^s, h^s$)
9: \hspace{1em} else
10: \hspace{2em} ITERATE*($G^z, Q^z, h^z$)
11: \textbf{until} $u \neq \text{NIL}$
12: Let $\mu$ be the time dependent costs from the $s$-$z$ path via node $u$

// Phase 2
13: \textbf{repeat}
14: \hspace{1em} if a certain selection rule holds then
15: \hspace{2em} ITERATE*($G^s, Q^s, h^s$)
16: \hspace{1em} else
17: \hspace{2em} ITERATE*($G^z, Q^z, h^z$)
18: \textbf{until} $\min\{d^s_z \mid v \in Q^z\} > \mu$

// Phase 3
19: \textbf{repeat}
20: \hspace{1em} Let $v \leftarrow \text{arg min} \{d^s_w + h^s_w \mid v \in Q^s\}$
21: \hspace{1em} $Q^s \leftarrow Q^s \setminus \{v\}$
22: \hspace{1em} if $v \in R^z$ then
23: \hspace{2em} \textbf{foreach} $(v, w) \in A^s$ do
24: \hspace{3em} \text{LABEL-UPDATE}(v, w, Q^s)
25: \hspace{1em} \textbf{until} $d^s_z < \infty \land z \notin Q^s$
D.4 Time dependent Contraction Hierarchies

Batz et al. [10, 11] extended the Contraction Hierarchies (see Section 6.4) method to fit in a time dependent environment. Recall that the Contraction Hierarchies preprocessing consists of two parts. First, a node ordering ‘<’ is determined and next, nodes are bypassed one-by-one based on that ordering. In the time-dependent situation, the same ordering of nodes is used as in the static case. This is motivated by Batz et al. by the assumption that, averaged over the planning period, the importance of nodes is not heavily affected by its exact traffic pattern. The second part, the bypassing of all nodes, is adapted to the time dependent environment as follows. Suppose node $v$ has to be bypassed. For every incoming arc $(u, v)$ and every outgoing arc $(v, w)$ it now should be checked whether the path $(u, (u, v), v, (v, w), w)$ might be a unique shortest path at any point in time. If so, the shortcut $(u, w)$ is added to the graph with the time dependent arc cost $c_{uw}(t)$ that equals $c_{uv}(t) + c_{vw}(t + c_{uw}(t))$.

In the static case the Contraction Hierarchies are exploited by Algorithm 6.8 where the forward search uses $G_\uparrow$ and the backward search uses $G_\downarrow$. The forward search is easy to generalize to the time dependent situation, since the departure time $t_0$ is known. Since the arrival time at $z$ is not known in advance, the backward search cannot be generalized that easily. Therefore, the following procedure is proposed by Batz et al. First, all nodes that can reach node $z$ in $G_\downarrow$ are determined. During this search all arcs in $A_\downarrow$ that connect nodes that can reach $z$ in $G_\downarrow$ are marked. The set of all marked arcs is $A_\downarrow' \subseteq A_\downarrow$. Now, a (time dependent) forward search is started at $t_0$ in the graph $(V, A_\uparrow \cup A_\downarrow')$. This procedure is correct for the same reasons as the static case: the way the nodes are bypassed ensures that there exists a shortest path $P$ that consists of two segments: the first segment is part of $G_\uparrow$ while the second is part of $G_\downarrow$. Since all arcs of $P$ are in $A_\uparrow \cup A_\downarrow'$, this path or some other shortest path will be found from the forward search.
Bibliography


Bibliography


286 Bibliography


Samenvatting

Het kortste-pad probleem op echte wegennetten
Theorie, algoritmen en berekeningen

Het kortste-pad probleem is een bekend probleem in de grafentheorie. Een graaf bestaat uit een verzameling knopen en gerichte takken. Een pad van een bronnknoop naar een doelknoop loopt via één of meerdere aaneengeschakelde takken in de graaf. Aan elke tak kunnen kosten worden toegekend. Die kosten kunnen bijvoorbeeld de lengte van een tak representeren of de tijdsduur die nodig is om over de tak te reizen. De kosten van een pad worden gedefinieerd door de som van de takkosten van de takken die onderdeel zijn van het pad. Het ‘kortste-pad probleem’ richt zich op het zoeken van een goedkoopste pad tussen een gegeven bron- en doelknoop. Afhankelijk van wat de takkosten representeren, krijgt het begrip ‘goedkoopst’ een specifieke betekenis: Wanneer de takkosten bijvoorbeeld lengtes voorstellen, dan betekent goedkoopst eigenlijk ‘kortst’. Wanneer de takkosten tijdsduren voorstellen, dan betekent het ‘snelst’. In deze samenvatting is gekozen om de term ‘goedkoopst’ te hanteren om minimale kosten aan te duiden. Om aansluiting met de literatuur te houden, wordt wel gesproken over het ‘kortste'-pad probleem en over ‘kortste pad’ algoritmen. Een wegennet is te modelleren als een graaf. De knopen representeren in dat geval afslagen, kruisingen en splitsingen, terwijl de wegsegmenten tussen deze knopen de takken voorstellen. De lengte van het wegsegment, ofwel de tijd die nodig is om het wegsegment te berijden, kan gekozen worden als kostenindicatie van een tak.

Al in de vijftiger jaren van de vorige eeuw werden verschillende algoritmen voor het oplossen van het kortste-pad probleem gepubliceerd. In 1959 heeft Dijkstra [59] een algoritme beschreven in slechts tweeënhalf pagina. Dit algoritme wordt tot op de dag van vandaag nog gebruikt, hetzij in zijn originele vorm, hetzij aangepast. Dit algoritme is een zogenaamd polynomiaal algoritme: de rekentijd wordt begrens door een polynomiale functie met als parameters het aantal knopen en het aantal takken. Het kortste-pad probleem is daarom in principe een niet moeilijk op te lossen probleem. De belangrijkste reden dat er nog altijd onderzoek gedaan wordt naar het kortste-pad probleem, is dat toepassing ervan wenselijk is op steeds omvangrijkere grafen, als gevolg van logistieke applicaties en het omvangrijke gebruik van navigatiesystemen. Daarnaast is het kortste-pad probleem vaak terug te vinden als onderdeel van een groter, meer complex probleem. Om tot een oplossing daarvan te kunnen komen, kan het nodig zijn het kortste-pad probleem meermaals op te lossen. Het is dan van belang dat dit zo snel en efficiënt mogelijk gebeurt.

Alle bekende kortste pad algoritmen zijn iteratief. In Dijkstra’s algoritme wordt, in elke iteratie een knoop e gekozen waarvoor een goedkoopste pad vanaf de bronnknoop
naar $v$ bekend is. Dat pad wordt definitief gemaakt. De kosten van dat pad worden aangeduid met het label $d_v$. Op basis van dat pad wordt geprobeerd om paden naar naburige knopen van $v$ te vinden of te verbeteren. Dit wordt het scannen van knoop $v$ genoemd. Een eenmaal gescande knoop komt in Dijkstra’s algoritme nooit meer terug. Dijkstra’s algoritme wordt daarom ook een ‘label-setting’ algoritme genoemd: Immers, in elke iteratie wordt het pad naar de gekozen knoop vastgelegd. Het bepalen van een knoop die gescand gaat worden in een iteratie, is een tijdrovende zoektocht. Daarom is er uitgebreid onderzoek gedaan naar efficiënte datastructuren om snel tot de juiste knoopprioriteit te komen. Andere onderzoekers hebben deze tijdrovende stap overslagen en kozen een willekeurige knoop in elke iteratie van hun algoritme. Het gevaar bestaat daarbij, dat er tijdens het uitvoeren van de algoritme op een later moment een goedkoper pad gevonden wordt naar een al eerder gescande knoop. In dat geval keert die knoop terug. Tijdens een iteratie wordt er dus geen pad vastgezet, maar worden goedkopere paden wel benut. Daarom worden zulke kortste pad algoritmen aangeduid met de term ‘label-correcting’. Het idee is dat er mogelijk weliswaar meer iteraties nodig zijn dan bij een label-setting algoritme, maar dat de kortere rekentijd per iteratie toch leidt tot een integraal snellere methode.

In situaties waarbij specifiek gezocht wordt naar het goedkoopste pad tussen een bronknoop en één enkele doelknoop, kan er sneller een oplossing gevonden worden. Het is bijvoorbeeld mogelijk om de zoektocht niet alleen voorwaarts vanuit de bronknoop, maar tegelijk ook achterwaarts vanuit de doelknoop te starten. Dit wordt bi-directioneel zoeken genoemd. De voorwaarts en achterwaarts doorzochte gebieden zullen elkaar op enig moment raken, waarbij verwacht wordt dat dit ergens tussen de bron- en doelknoop gebeurt. Wanneer beide gebieden elkaar raken betekent dit twee dingen: Er is een (koppel)knoop gevonden waarvoor het goedkoopste pad van de bronknoop tot die knoop bepaald is. Tegelijk is dan het goedkoopste pad van die knoop tot de doelknoop gevonden. Het is al langer bekend dat een goedkoopste pad tussen de bronknoop en een doelknoop niet altijd via de koppelknoop hoeft te lopen. Er is echter een snelle methode bekend waarmee, direct na het vinden van zo’n koppelknoop, eenvoudig een goedkoopste pad tussen bron- en doelknoop bepaald kan worden. Er hoeven geen voorwaarts of achterwaarts iteraties meer gedaan te worden en men kan overgaan naar een zogeheten post-processing stap.

In het onderzoeksgebied van de kunstmatige intelligentie is een generiek algoritme beschreven met de naam A*. Hierbij wordt in een iteratie niet gekozen voor een knoop waarbij de kosten van het pad vanaf de bronknoop tot deze knoop minimaal zijn. Gekozen wordt die knoop, waarbij de som van de kosten van dat pad en een schatting van de kosten van het pad vanaf de knoop tot de doelknoop, minimaal zijn. Zo wordt de zoektocht in de richting van de doelknoop gestuurd. Voor de gebruikte schatting wordt vaak de hemelsbrede afstand gebruikt. Als er specifiek gezocht wordt naar een snelste route, dan wordt verondersteld dat je hemelsbreed kunt rijden met de maximaal toegestane snelheid binnen de graaf. Op deze manier worden de daadwerkelijke kosten van het resterende pad naar de doelknoop altijd onderschat. Al snel werd onderzoek gedaan naar de mogelijkheid om ook een schatter te gebruiken in bi-directionele methoden. Het bleek echter lastig om aan te geven wanneer het algoritme dan kan stoppen. De hierboven genoemde post-processing
deel niet gebruikt te kunnen worden. Het voorwaarts en achterwaarts zoeken moest dus langer worden voortgezet. In 1994 werd een schatter voorgesteld waarbij tijdens het voorwaartse zoeken niet alleen de schatting van de kosten naar de doelknoop wordt meegenomen, maar ook de schatting voor de kosten terug naar de bronknoop. Bij het achterwaartse zoeken wordt niet alleen de schatting van de kosten naar de bronknoop, maar ook de schatting van de kosten terug naar de doelknoop meege- nomen. Dit worden gebalanceerde schatters genoemd. Voor elke knoop geldt dat de som van de voorwaartse en de achterwaartse schatter een constante is (meestal nul). Het voordeel van gebalanceerde schatters is dat deze kunnen worden inge- bed in een bi-directionele variant van Dijkstra’s algoritme, door slechts de takkosten aan te passen op basis van de schatters. Het gevolg hiervan is dat het zoekproces in de voorwaartse en in de achterwaartse richting gestopt kan worden zodra zij elkaar raken. Alleen de eerder genoemde post-processing stap moet dan nog uitgevoerd worden. Vanaf 1994 worden in de literatuur voornamelijk zulke gebalanceerde schatters gebruikt.

Wanneer bij het voorwaarts zoeken alleen de schatting van de kosten naar de doel- knoop wordt gebruikt en bij het achterwaarts zoeken alleen de schatting van de kosten naar de bronknoop, dan worden deze schatters aangeduid met de term ‘sym- metrisch’ in dit proefschrift. Het gebruik van symmetrische schatters wordt welis- waar nog regelmatig genoemd in de literatuur, maar er wordt al snel bij geschreven dat de voorkeur naar gebalanceerde schatters uitgaat. Dit omdat er dan tenmin- ste een duidelijke stop conditie is: het elkaar raken van beide zoek richtingen. Bij symmetrische schatters is dat niet zo. Het resterende werk na het vinden van een koppelknoop wordt drastisch verkort middels een in dit proefschrift beschreven methode. Deze methode is zowel toepasbaar op symmetrische als op gebalanceerde schatters. Bij gebalanceerde schatters blijkt dat de gangbare post-processing stap gewoon opgenomen wordt in het algoritme zelf. Als de symmetrische schatter is gebaseerd op de hemelsbrede afstand, blijkt tevens dat de rekentijd en het door- zochte gebied slechts licht verkleind worden. De nieuw voorgestelde methode, die gezien kan worden als een generalisatie van de huidige gangbare post-processing stap, heeft dan ook voornamelijk een theoretisch belang. Bij het gebruik van een hemels- brede schatter blijken de beide zoekrichtingen elkaar eerder te raken wanneer er een symmetrische schatter wordt gebruikt dan wanneer er een gebalanceerde schatter gebruikt wordt. Door het daarna resterende werk te verdelen over de voorwaartse en de achterwaartse richtingen middels zogeheten scalaire projecties, blijkt het mogelij- ki te zijn om het totale doorzochte gebied te verkleinen en de benodigde rekentijd drievoudig te verkorten.

De laatste decennia is, mede door de opkomst van navigatiesystemen, het optimali- seren in zeer grote grafen gewenst geworden. Het gebruik van Dijkstra’s algoritme leidt dan tot onacceptabele rekentijden. Als op dezelfde graaf vaker een kortste-pad probleem moet worden opgelost, kan het zinvol zijn om vooraf extra data aan de graaf toe te voegen. De kortste pad berekeningen kunnen hier later gebruik van maken. Zo wordt de oplossingsruimte flink verkleind, waardoor de berekeningen versneld uitgevoerd kunnen worden. Het toevoegen van extra data kan op verschillende manieren. Een van deze manieren maakt gebruik van zogenaamde wegwijzers.
Samenvatting

Een klein aantal knopen in de graaf wordt aangewezen als wegwijzer. De kosten van de goedkoopste paden van elke knoop naar elke wegwijzer worden éénmalig uitgerekend. Dat geldt ook voor de kosten van de goedkoopste paden van elke wegwijzer naar elke knoop. Op basis van deze vooraf uitgerekende kosten kan eenvoudig een zeer goede schatter bepaald worden voor verder gebruik in een A* algoritme. Uit de in dit proefschrift beschreven experimenten blijkt dat van een groot aantal takken de aangepaste takkosten nul zijn, wanneer een schatter op basis van wegwijzers gebruikt wordt. Dit kan nadelige gevolgen hebben voor de efficiëntie van het algoritme. Voor een groot aantal knopen kunnen de (aangepaste) kosten van het pad van de bronknoop tot die knoop gelijk zijn. De kans bestaat dat er, bij een minder geschikte keuze uit deze knopen, relatief lang in de verkeerde richting gezocht wordt. In dit proefschrift wordt voorgesteld om eerst een zogenaamde Depth-First-Search wandeling te maken door de omgekeerde graaf over uitsluitend takken waarvoor de aangepaste kosten nul zijn. Hierdoor wordt de totale rekentijd voor uni-directionele implementaties aanzienlijk verkort. Ook blijken de verschillende label-setting implementaties dan een min of meer gelijk aantal iteraties nodig te hebben. Dit is bij label-setting methoden ook te verwachten. Bij schattingen die gebaseerd zijn op wegwijzers blijkt dat het gebruik van symmetrische schatters ertoe leidt dat de beide zoekrichtingen elkaar relatief laat raken. Dit komt vermoedelijk door het grote aantal takken waarvoor de aangepaste kosten nul is.

Niet altijd leidt een bi-directionele methode tot een verbetering van de rekentijd ten opzichte van een uni-directionele methode. Een van de beschreven pre-processing technieken probeert de graaf zodanig te partitioneren dat het aantal grensknopen minimaal is. Een grensknoop is een knoop die op de grens van twee cellen in die partitie ligt. Deze grenzen zijn onafhankelijk van de gebruikte takkosten. Bij gegeven takkosten worden er shortcuts bepaald tussen elk tweetal grensknopen van één cel. De kosten van zo’n shortcut zijn gelijk aan de kosten van het goedkoopste pad tussen de twee grensknopen, waarbij alleen via knopen van de cel zelf gereisd mag worden. Omdat het aantal knopen per cel klein is, zijn deze shortcuts snel te bepalen. Dit type pre-processing wordt hybride genoemd. In de eerste stap wordt de graaf gepartitioneerd, waarbij alleen naar het bestaan van de takken gekeken wordt. In de tweede stap worden de shortcuts berekend op basis van een gegeven set met takkosten. Omdat het aantal knopen in een cel vaak fors groter is dan het totaal aantal grensknopen in de partitie, blijkt dat bi-directionele methoden bij dit type pre-processing minder geschikt zijn dan uni-directionele methoden. Juist bij dit type pre-processing is de initiële Depth-First-Search wandeling van toegevoegde waarde.

Deze samenvatting wordt afgesloten met een korte beschrijving van de thematiek van ieder hoofdstuk. In hoofdstuk 1 wordt een algemene inleiding gegeven. Dit bevat een globaal beeld van het onderzoek naar het kortste-pad probleem zoals dit met name de afgelopen 60 jaar heeft plaatsgevonden. Hoofdstuk 2 bevat de benodigde wiskundige notaties en definities om het kortste-pad probleem op een formele wijze te beschrijven. In hoofdstuk 3 wordt een uitgebreid overzicht gegeven van algoritmen uit de literatuur waarmee het kortste-pad probleem kan worden opgelost. Hierbij wordt gebruik gemaakt van één generiek algoritme, waarvan de verschillende algoritmen een specifieke instantie zijn. Bij de beschreven algoritmen wordt de complexiteit
Samenvatting

Het onderscheid tussen label-setting en label-correcting methoden komt uitgebreid aan de orde. Het betreft hier algoritmen die gebruikt kunnen worden om het goedkoopste pad van één bronknoop naar één of meerdere doelknopen te bepalen. Wanneer er sprake is van slechts één doelknoop kan er ook bi-directioneel gezocht worden. In hoofdstuk 4 wordt de bestaande literatuur over dit onderwerp samengevat. Nieuw is dat hier een algoritme wordt gepresenteerd waarmee bi-directioneel zoeken toegepast kan worden in label-correcting methoden. Hoofdstuk 5 beschrijft hoe een schatting van de nog af te leggen afstand gebruikt kan worden om een goedkoopste pad tussen twee locaties sneller te kunnen berekenen. Ook wordt beschreven hoe zo'n schatter gebruikt kan worden in een bi-directionele omgeving. In dit hoofdstuk komt het gebruik van symmetrische schatters naar voren met daarbij nieuwe theorie waarmee het werk, dat nog moet gebeuren zodra het voorwaartse en achterwaartse zoeken elkaar raken, sterk verkort kan worden. Ook komt hier aan de orde hoe het werk verdeeld kan worden over beide zoekrichtingen middels scalaire projecties. In hoofdstuk 6 worden verschillende pre-processing technieken beschreven, waarbij onderscheid gemaakt wordt tussen statische en dynamische technieken. Statische technieken zijn onafhankelijk van de takkosten en dynamische technieken zijn er juist op gebaseerd. Hier komen ook de eerder genoemde wegwijzers en de hybride pre-processing techniek aan de orde. In hoofdstuk 7 wordt een uitgebreide experimentele evaluatie beschreven. In de appendices wordt ingezoomd op datastructuren, het specifieke kortste-pad probleem waar de goedkoopste paden van elke knoop naar elke knoop bepaald moet worden en het gebruik van tijdsafhankelijke takkosten.
Summary

The Shortest Path Problem is a well-known problem in Graph Theory. A graph consists of a set of nodes and directed arcs. A path from a source node to a target node goes via one or more arcs that are connected. A cost can be assigned to each arc. Such a cost for instance represents the length of the arc, or the time needed to travel along the arc. The cost of a path is defined as the sum of the arc costs of each arc that is included in the path. The ‘Shortest Path Problem’ addresses how to determine a cheapest path from any given source node to a specified target node. Depending on the arc costs that are provided a ‘cheapest’ path has a certain meaning. If the arc costs represent arc lengths, a ‘shortest’ path has to be found. In case the arc costs represent travel times, a ‘fastest’ path has to be determined. In this summary, we use the the term ‘cheapest’ to denote minimal costs. In order to stay in line with related literature, we write ‘Shortest’ Path Problem and ‘shortest’ path algorithms. A roadmap can be modeled as a graph. The nodes represent either exits, crossings or intersections and the road segments between those nodes are represented by the arcs. The cost of an arc can be expressed as either the length of that road segment, or the time needed to travel that particular road segment.

Already in the 1950s, several algorithms for solving the Shortest Path Problem were published. In 1959, Dijkstra [59] described an algorithm in only two and a half pages. This algorithm is still in use today, both modified and in its original form. This algorithm is a so-called polynomial algorithm: the running time is bounded by a polynomial function, the parameters of which consist of the number of nodes and the number of arcs. Hence, the Shortest Path Problem is not that difficult to solve in its basic form. The main reason for the Shortest Path Problem to remain an interesting research subject is that the size of the graphs considered has increased drastically over time, mainly due to the demand from applications in logistics and the extensive use of navigation systems. In addition, the Shortest Path Problem frequently plays a role in other, more complicated problems. To solve such a problem, the Shortest Path Problem frequently has to be solved a number of times. It is important in such cases to be able to solve these Shortest Path Problems as fast as possible.

All known shortest path algorithms are iterative. In each iteration of Dijkstra’s algorithm, a node $v$ is selected for which a cheapest path from the source node to that node $v$ is known. This path is now considered to be definitive. The cost of that path is denoted by a label $d_v$. Based on that path, an attempt is made to find new or improved paths towards neighbors of node $v$. This is called the ‘scanning’ of node $v$. Once a node is scanned, it will never return in later iterations of Dijkstra’s algorithm. Therefore, Dijkstra’s algorithm is called a ‘label-setting’ algorithm: after each iteration, the path to a selected node is definitive. The search for a node that is about to be scanned during an iteration, is time consuming. In-depth research has therefore been done after efficient data structures that quickly achieve the correct
node selection. Other researchers skip this elaborate step and simply choose a node arbitrarily in each iteration of their algorithm. The potential drawback of this approach is that, during algorithm execution in a later stage, a cheaper path may be found to a previously selected node. In other words, the path to that node will be improved. In such a case, the node can be re-selected in a later iteration. No path becomes definitive during a specific iteration, but improved paths will be used during each iteration. This kind of shortest path algorithm is therefore called ‘label-correcting’. The idea is that, most likely, more iterations have to be executed than with a label-setting algorithm, but that the reduction in the execution time of each iteration will lead to an overall faster algorithm.

In situations where a cheapest path has to be determined between a source node and a single destination node, it is possible to reduce the computing time. For example, one might not only search forward from the source node towards the destination node, but simultaneously calculate that route backwards from the target node to the source node. This is called bidirectional search. The forwards and backwards explored areas will eventually meet somewhere between the source node and the destination node. When both explored areas meet, this means at least two things: a (connecting) node is found for which a cheapest path from the source node to that node is known. Additionally, a cheapest path from this particular connecting node to the destination node is revealed. It is a known fact that a cheapest path from the source node to the destination node will not necessarily pass this connecting node. Yet, a fast method exists to determine a cheapest path between the source node and the destination node as soon as the explored areas intersect. The forward and backward iterations can be stopped immediately and only a so-called post-processing step remains to be executed.

In Artificial Intelligence research, a generic algorithm is presented under the name $A^*$. Instead of selecting a node for which the cost of the path from the source node to that node is minimal, a node is selected for which the sum of the cost of that path and the estimated cost of the path from that node to the target node, is minimal. In this way, the search is directed towards the destination node. A commonly used estimator here, is the way the crow flies. If one specifically searches for the fastest path, the estimate is based on the assumption that the distance can be traveled along a straight line with the maximum speed allowed within the entire graph. Such a straight line-based estimate is always a lower bound to the cost of the cheapest path to the target node. Actual costs of the path to the destination node will thus remain underestimated. Research followed to incorporate such estimators in bidirectional methods as well. It turns out to be difficult to provide an accurate stop criterion. The abovementioned post-processing step seemed unfit for direct use when estimators are involved. Therefore, the iterations of the forward and backward search had to continue. In 1994, a method was proposed with an estimator where, during the forward search not only costs towards the destination node were taken into account, but also the estimated costs back towards the source node. During the backward search, both the estimate towards the source node as well as the estimate back to the destination node are taken into account. This is called a balanced approach. For each node, the sum of the forward and backward estimates is a constant (usually
The advantage of a balanced approach is that the estimator can be embedded in a bidirectional version of Dijkstra's algorithm, simply by modifying the arc costs based on the estimators. Thus, the forward and backward searches can cease as soon as they intersect. Only the aforementioned post-processing step still has to be executed. Since 1994, most literature contains methods that use such a balanced approach.

In this thesis we use the term 'symmetric approach' for the situation where in the forward search only estimates towards the target node are used, and in the backward search only estimates towards the source node are used. Although symmetric approaches are still mentioned in related literature, usually a balanced approach is preferred due to the clear stop condition: stop as soon both search spaces intersect. In a symmetric approach, such a stop condition cannot be used. The remaining actions once a connecting node is found, can drastically be reduced with the method described in this thesis. This method can be used with both symmetrical as well as with balanced estimators. In the latter case, the post-processing step is simply incorporated in the algorithm itself. It turns out that in a symmetric approach, both the searched areas and the computing time are slightly reduced if an estimator is used that is based on the straight line distances. The proposed, new method, which can be considered a generalisation of the currently regular post-processing step, is particularly interesting from a theoretical point of view. If the estimates are based on straight line distances, the forward and backward search spaces seem to intersect sooner when a symmetric approach is used, compared to the use of a balanced approach. By subsequently dividing the remaining work over forward and backward iterations, using so-called scalar projections, it turns out that it is possible to decrease both the total searched area as well as the running time of the algorithm.

Over the past few decades, due to developments such as the introduction of car navigation systems, the use of considerably large graphs became increasingly common. If the same graph is used for solving many shortest path problems, it might be fruitful to include additional data. The shortest path calculations will use this data in a later stage. Thus, the solution space can be reduced drastically and consequently the calculations will execute much faster as well. Adding data can be done in a variety of methods. One way to add some data to the graph is the selection of so-called landmarks. A small number of nodes in the graph are declared to be landmarks. The cost of the cheapest paths from any node to any landmark, are determined and stored. The same is done for the cheapest path costs from any landmark to any node. Based on these pre-calculated costs, one can derive a very strong estimator, which can subsequently be used in an A* algorithm. The experiments presented in this thesis show that a landmark-based estimator has a specific property where, for a large number of arcs in the graph, the modified arc cost based on this estimator is zero. This may affect the efficiency of the algorithm. While selecting a node in each iteration for which the modified costs of the path from the source node to that node is minimal, these costs may turn out to be the same for a large number of nodes. Hence, it might happen that it takes a relatively long time before the search continues in the proper direction. In this thesis it is proposed to start by running a so-called Depth-First-Search traversal in the reverse graph, using only...
arcs with modified arc costs equal to zero. This allows the cumulative running time for unidirectional algorithms that use a landmark-based estimator, to be reduced significantly. Moreover, the number of iterations needed by different label-setting algorithms are now more or less equal, which is the expected behavior of label-setting methods. If landmark-based estimators are used in a symmetric approach, it turns out that it takes relatively (too) long before both search directions intersect. This is suspected to be the result of the large number of arcs with modified arc cost nil.

A bidirectional method cannot always compete successfully with a unidirectional method. One of the preprocessing techniques described tries to partition the graph in such a way that the number of border nodes is minimized. A border node is a node on the border of two cells of the partition. These border nodes are determined independently of the actual arcs costs. Based on a specific set of arc costs, shortcuts are determined between any pair of border nodes of a cell. Such a shortcut represents a cheapest path between the border nodes, which goes via nodes of that particular cell only. Since the number of nodes in each cell is small, these shortcuts can be determined rather quickly. This type of preprocessing is called ‘hybrid’. In the first step, the graph is partitioned, taking into account only the existence of arcs. In the second step, the actual arc costs are used to determine the shortcuts based on a given set of arc costs. Since the number of nodes in a cell is larger than the total number of border nodes in the partition, it turns out that a bidirectional approach is less effective than a unidirectional approach. Specifically for this type of preprocessing, the initial Depth-First-Search traversal is a valuable and important addition.

We conclude this summary with a short outline of each chapter. Chapter 1 contains a general introduction. The research of the past 60 years is briefly summarised there. In Chapter 2, the notation and definitions needed to formally describe the Shortest Path Problem, are presented. In Chapter 3, a detailed overview of existing algorithms in relevant literature is presented. One generic algorithm is presented of which all other algorithms are a specific case. The complexity of each algorithm is elaborated on and the difference between label-setting and label-correcting algorithms is described in detail. The algorithms mentioned in this chapter can be used to solve the shortest path problem from one source node to a set of destination nodes. In case there is only one specific target node, bidirectional search can be used. In Chapter 4, the existing literature on this subject is summarized. A new algorithm that makes it possible to add bidirectional search to label-correcting methods, is presented as well. In Chapter 5, a description follows of how an estimate of the cost of the remaining path can be used to decrease the computing time. Furthermore, it is described how estimators can be used in a bidirectional setting. In this chapter, the symmetric approach is presented with a new theory to reduce the remaining work after the search spaces intersect. It is explained how that work can be reduced over forward and backward iterations, based on scalar projections. In Chapter 6, several preprocessing techniques are described. A distinction is made between static and dynamic techniques. Static preprocessing techniques are independent of the arc costs, while dynamic preprocessing techniques explicitly use them. Here, the aforementioned landmarks and hybrid preprocessing are described as well. Finally, Chapter 7 contains an extensive experimental evaluation. In the appendices, some
topics are described in more detail: the specific shortest path problem where the cheapest paths between all pairs of nodes have to be determined, data structures and the use of time dependent arc costs.
Curriculum Vitae

Henk Post was born January 4th, 1972 in Delft. In 1990, he finished his pre-university education (VWO) at Guido de Brès in Rotterdam. The same year he started to study Mathematics at Delft University of Technology. In 1995, he wrote a Master’s thesis on a real-time Pickup and Delivery Problem with Time Windows.

After completing this thesis, he implemented a solution for the Pickup and Delivery problem with Future Technology B.V. Some main areas of his work are the Shortest Path Problem, the Vehicle Routing Problem and various problems in real-time environments. In 2000, Future Technology was acquired by Connexxion. Since then, Henk also works on issues that arise in the areas of public transportation and emergency vehicle routing.

From 1999 onwards, Henk is a guest lecturer at Delft University of Technology on ‘Transport, Routing- and Scheduling problems’. He has supervised five master students.

Henk started working on his PhD thesis in 2006. He still works as a solution architect at Connexxion.