# Fastest Paths, Almost Disjoint Paths, and Beyond

**Tim Bergner**

TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Vom Fachbereich Mathematik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades
Doktor der Naturwissenschaften
(Doctor rerum naturalium, Dr. rer. nat.)
genehmigte Dissertation

| | |
|---|---|
| Gutachter: | Prof. Dr. Sven Oliver Krumke |
| | Prof. Dr. Sebastian Stiller |

Datum der Disputation: 07.10.2022

DE-386

# Abstract

This thesis is primarily motivated by a project with Deutsche Bahn about offer preparation in rail freight transport. At its core, a customer should be offered three train paths to choose from in response to a freight train request. As part of this cooperation with DB Netz AG, we investigated how to compute these train paths efficiently. They should be all "good" but also "as different as possible". We solved this practical problem using combinatorial optimization techniques. In particular, we formulated it as a multi-criteria shortest paths problem in a time-dependent network with various constraints.

At the beginning of this thesis, we describe the practical aspects of our research collaboration in detail. This includes, for example, the data provided by Deutsche Bahn, our mathematical models, the algorithmic ideas that we exploit, problems we encountered and our approaches to solve them. The more theoretical problems, which we consider afterwards, are divided into two parts.

In Part I, we deal with a dual pair of problems on directed graphs with two designated end-vertices. The *Almost Disjoint Paths* (ADP) problem asks for a maximum number of paths between the end-vertices any two of which have at most one arc in common. In comparison, for the *Separating by Forbidden Pairs* (SFP) problem we have to select as few arc pairs as possible such that every path between the end-vertices contains both arcs of a chosen pair. The main results of this more theoretical part are the classifications of ADP as an **NP**-complete and SFP as a $\mathbf{\Sigma}_2^p$-complete problem.

In Part II, we address a simplified version of the practical project: the *Fastest Path with Time Profiles and Waiting* (FPTPW) problem. In a directed acyclic graph with durations on the arcs and time windows at the vertices, we search for a fastest path from a source to a target vertex. We are only allowed to be at a vertex within its time windows, and we are only allowed to wait at specified vertices. After introducing *departure-duration functions* we develop solution algorithms based on these. We consider special cases that significantly reduce the complexity or are of practical relevance. Furthermore, we show that already this simplified problem is in general **NP**-hard and investigate the complexity status more closely.

# Zusammenfassung

Diese Arbeit ist durch ein Forschungsprojekt mit der Deutschen Bahn zur Angebotserstellung im Schienengüterverkehr motiviert. Das Ziel dieses Projektes war es, Kunden für eine Güterzuganfrage im Gelegenheitsverkehr drei verschiedene Angebote machen zu können. Im Rahmen der Zusammenarbeit mit der DB Netz AG haben wir untersucht, wie man diese Trassen effizient berechnen kann. Diese sollten alle „gut“ aber auch „möglichst verschieden“ sein. Gelöst haben wir dieses Problem mit Techniken der kombinatorischen Optimierung. Insbesondere haben wir es dafür als mehr-kriterielles kürzeste Wege Problem in einem zeitabhängigen Netzwerk unter diversen Nebenbedingungen formuliert.

Zu Beginn dieser Arbeit beschreiben wir die praktischen Aspekte des Forschungsprojektes. Dazu gehören zum Beispiel die von der Deutschen Bahn bereitgestellten Daten, unsere mathematischen Modelle, die von uns entwickelten algorithmischen Ideen, aufgetretene Probleme und unsere Lösungsansätze dafür. Die theoretischen Probleme, welche wir anschließend betrachten, gliedern wir in zwei Teile.

In Teil I untersuchen wir zwei duale Probleme auf gerichteten Graphen mit zwei ausgewiesenen Endknoten. Im *Almost Disjoint Paths* (ADP) Problem suchen wir möglichst viele Pfade zwischen den Endknoten, die paarweise maximal eine Kante gemeinsam haben. Für das *Separating by Forbidden Pairs* (SFP) Problem sollen möglichst wenige Kantenpaare gewählt werden, so dass jeder Pfad zwischen den Endknoten beide Kanten eines solchen Paares enthält. Die Hauptergebnisse dieses eher theoretischen Teils sind die Klassifizierungen von ADP als **NP**-vollständiges und SFP als $\boldsymbol{\Sigma}_2^p$-vollständiges Problem.

In Teil II behandeln wir eine vereinfachte Version des praktischen Problems aus dem Forschungsprojekt: das *Fastest Path with Time Profiles and Waiting* (FPTPW) Problem. In einem gerichteten, azyklischen Graphen mit Laufzeiten auf den Kanten und Zeitfenstern an den Knoten suchen wir einen schnellsten Weg von einem Start- zu einem Zielknoten. Dabei dürfen wir uns an jedem Knoten nur innerhalb von dessen Zeitfenstern aufhalten und nur an bestimmten Knoten warten. Wir führen *Departure-Duration Functions* ein und entwickeln darauf basierende Lösungsalgorithmen. Neben der Betrachtung einiger Spezialfälle, welche die Komplexität signifikant reduzieren oder praktisch relevant sind, zeigen wir auch, dass schon dieses vereinfachte Problem im Allgemeinen **NP**-schwer ist und analysieren die Komplexität dieses Problems genauer.

# Acknowledgments

A lot of people have contributed to making this thesis what it is. At this point, I would like to explicitly thank at least some of them. First of all, of course, there is my supervisor Prof. Dr. Sven O. Krumke. Sven, thank you for giving me this opportunity, for allowing me a lot of freedom, and for supporting me whenever I was in need of it. What always impresses me is your almost infinite knowledge of literature, and what I particularly appreciate is your unbelievable abundance of ideas.

I would also like to thank Oliver Bachtler with whom I spent many hours and days doing math together. Oliver, thanks for the great cooperation, for smart ideas from your side, and for listening to my thoughts. I very much appreciate that you are spontaneous and always there to discuss problems, algorithms, or ideas. It is a lot of fun to work with you, and especially Part I of this thesis has gained tremendously from working with you.

Particularly direct improvements to this work came from all my proofreaders. For this extremely helpful feedback I would like to thank Oliver Bachtler, Tobias Dietz, Irene Heinrich, Christoph Hertrich, Steffen Plunder, Eva Schmidt, Manuel Streicher, Ingo Tragelehn, and Thomas Ullmert. I am incredibly grateful to you, thank you all very much!

Furthermore, I would like to thank Ingo Tragelehn and Dr. Oliver Kolb from Deutsche Bahn, with whom we worked on the practical project. I enjoyed our collaboration very much. It was the foundation from which this thesis grew. Both our project and your support was very valuable. Thank you!

Finally, I would of course like to thank my family and friends who have always supported me throughout my studies and doctorate. It is always great to come home and just feel comfortable there!

# Contents

# Chapter 1

# Introduction

One of the greatest challenges of our time is the transformation to a more sustainable society. The transport sector plays a crucial role in this process and an important task in this context is to get more goods onto the rails. For this purpose, rail freight transport must become more attractive. Furthermore, as the railway infrastructure is limited, we must make the best use of the available capacities.

Since 2019, the German railway operator Deutsche Bahn simplifies and improves short-term train path requests with the web application “Click & Ride”. After specifying train characteristics and route information customers are offered an automatically constructed train path within a few minutes. Before this service was available, all train paths for customer requests were constructed manually, which took considerably longer. So far, the web application only provides a single train path for each customer request, however.

A research cooperation between Deutsche Bahn and TU Kaiserslautern aims to improve and to diversify train paths offered to short-term train path requests. More precisely, the goal of this project was to develop and evaluate methods to algorithmically create multiple train paths, which should be “as different as possible” in order to offer each customer a wide range of options. In addition, we respected the utilization of track sections during the train path construction. From the customer’s point of view, train paths with low utilization are an advantage as the risk of delay is much lower. On the other hand, railway operators and other customers benefit from this as well because it results in a more evenly loaded rail network.

This thesis is motivated by this collaboration but apart from a brief outline of our practical attainments in Chapter 3, we mainly focus on more theoretical problems related to and inspired by this project. The treatment of these theoretical problems is divided into two parts. In Part I, we deal with the almost disjoint paths and separating by forbidden pairs problems. We formally introduce them, analyze their relation, and resolve their complexity statuses. In Part II, we focus on the fastest path with time profiles and waiting problem. We examine the temporal component of the problem, derive solution algorithms, and identify classes of instances on which we can find fastest paths in polynomial time.

## Part I: Almost Disjoint Paths and Forbidden Pairs

We consider the Almost Disjoint Paths (ADP) and Separating by Forbidden Pairs (SFP) problems, which have not been considered before and both of which we introduce. They belong to the field of graph theory, and they are inspired by both common practical and well-known theoretical problems. An instance of either problem is given by a directed graph with a source and a target vertex.

In the almost disjoint paths problem ADP we are looking for a set of paths from the source to the target from which every two paths are almost disjoint, meaning that every two paths are only allowed to have at most one arc in common. It exhibits parallels to the maximum flow problem and to the problem of finding disjoint paths, whose disjointedness requirement it relaxes in a natural and simple way. By requiring the paths to be almost disjoint we have an elementary criterion formalizing that we want sufficiently dissimilar paths. This does not only fit to the setting in our railway project where customer requests should be answered with various different train paths. Similar requirements also apply to other practical problems, for example in the construction of alternative routes.

In the separating by forbidden pairs problem SFP we select pairs of arcs such that every path from the source to the target contains both arcs of at least one chosen pair. As such, it is reminiscent of the minimum cut problem, where instead of choosing one arc, we now have to choose a pair of arcs on each path. In addition, the SFP adds another level on top of the already well-researched path avoiding forbidden pairs problem. This brings us back full circle to our practical project: with the latter problem we can model the requirement that train paths must not contain any cycles.

Our newly introduced problems have conceptual and content similarities to the maximum flow and to the minimum cut problem. It is therefore not surprising that also the almost disjoint paths and the separating by forbidden pairs problems form a dual pair. Unlike the maximum flow and the minimum cut problems, however, we have only weak duality for our problems. Another difference concerns the complexity of these problems, the study and classification of which we focus on in this thesis.

We deal with ADP and SFP in the first part of this thesis, and divide our results into three chapters. First, in Chapter 4, we formally introduce ADP and SFP, relate them to well-known problems from the literature, and focus on their relation to each other. With the help of integer programming formulations and suitable relaxations we prove that the problems form a dual pair. We construct a class of examples for which their duality gap is unbounded, which then, in particular, shows that ADP and SFP are only weakly dual. However, we show that strong duality holds if we restrict the graph class to those graphs that have a cut with a single outgoing arc. In this case, the problems are not only strongly dual but also polynomial time solvable.

Thereafter, in Chapters 5 and 6, we look at the problems ADP and SFP separately, where we mainly focus on their complexity. Starting with ADP in Chapter 5, we develop a dynamic program that, given a natural number $k$, allows us to compute up to $k$ almost disjoint paths on directed acyclic graphs. Moreover, we extend this approach so that it also works for any directed graph. We obtain that we can solve the almost disjoint paths problem in polynomial time when assuming this parameter $k$ to be constant. In general, however, ADP is **NP**-complete, which we prove by a reduction from the independent set problem. Since all graphs that we construct for this are acyclic, the almost disjoint paths problem remains **NP**-complete if we restrict the graph class to directed acyclic graphs.

In Chapter 6, we address the separating by forbidden pairs problem and, especially, its complexity. The main result of this chapter classifies SFP as $\mathbf{\Sigma}_2^p$-complete, which we prove by a reduction from the quantified satisfiability problem with two alternations $\mathbf{\Sigma}_2$SAT.

## Part II: Fastest Paths with Time Profiles and Waiting

The third problem that we introduce and consider in this thesis is the Fastest Path with Time Profiles and Waiting (FPTPW) problem. It is much closer to the train path construction from the practical project than the previously described problems. For this reason, we consider our fastest paths problem from a more algorithmic point of view. Instead of focusing on the complexity of the problem itself (as we do for ADP and SFP) we mainly develop an algorithm whose running time we analyze.

An instance of the fastest path with time profiles and waiting problem consists of a directed acyclic graph. Every arc has a duration, and every vertex contains time windows during which we are allowed to be there. In addition, the instance specifies for each vertex whether we are allowed to stop and wait there or whether we have to go straight through. For given source and target vertices, the task is to find a fastest path that complies with all additional requirements. Hence, a solution to this problem is a path from the source to the target that is enriched with departure times at the vertices it contains.

Problems similar to FPTPW occur not only in railway routing but also in the context of vehicle scheduling. In this area, however, we usually have slightly different conditions for the feasibility of solutions regarding waiting. The main reason for this is that cars can be parked almost anywhere without problems and in doing so they block much less infrastructure than trains do. But even if we only compare railway routing problems with each other, many differences come to light. The way in which the infrastructure, the time dependencies, and the feasible train paths are represented varies greatly. One reason for this is that managing and operating railroad traffic requires an enormous amount of effort. Due to very limited infrastructure, restricted maneuverability, and comparatively long breaking distances of trains, very precise timetables have to be drawn up in advance. To guarantee safety and to coordinate the trains, a lot of restrictions have to be considered. Thus, the nature of a railway routing problem highly depends on which components are already precomputed and available as input and which restrictions have to be ensured within the problem. Therefore, the theoretical problems that are considered in this context exhibit fundamental differences.

We deal with our fastest path with time profiles and waiting problem in Part II of this thesis. Basically, the task is to find a fastest path, that is, one with minimum duration. What distinguishes our situation from several similar problems is that we determine the duration of a path based on its actual departure and arrival times and not in relation to a fixed time point (for example an earliest start time). Therefore, we cannot easily adapt standard shortest path algorithms to FPTPW. Instead, we develop an algorithm that propagates functions as labels in Chapter 10. Such a function is associated to a specific vertex. It maps the possible departure times to the shortest durations to get there, which is why we call them *departure-duration functions*. In Chapter 9, we introduce these functions and analyze their structure. A key feature for their use in our fastest path algorithms is their piecewise linearity. Furthermore, we classify the breakpoints of the departure-duration functions and connect these to the paths that represent feasible solutions. These connections allow us to prove in Chapter 11 that our fastest path algorithm requires only polynomial time for certain classes of instances.

By taking a step back, we can ask the question of whether a valid path exists at all. To answer that question, we do not even need the departure-duration functions yet. A method that can provably deny this question restricts all time windows at the vertices to those time points that are used by any valid solution. This procedure, which we develop in Chapter 8, automatically synchronizes all the time windows. Since even this simpler problem is **NP**-complete, it is not surprising that this algorithm might require exponential time. However, if waiting is allowed at every vertex, the synchronization can be performed in polynomial time, and also our shortest path algorithm remains in this complexity class. Finally, we collect and refine all our complexity-related results about FPTPW in Chapter 12.

## Literature and Publications

Because this thesis contains quite different subjects, we refrain from summarizing related literature at this point and refer to Sections 3.3, 4.3 and 7.2 for detailed literature information tailored to the respective topics.

A large portion of Part I concerning the almost disjoint paths and the separating by forbidden pairs problem is joint work with Oliver Bachtler and Sven O. Krumke. It is already published in [BBK22].

Chapter 2

# Mathematical Preliminaries

*In this chapter, we list relevant concepts and introduce the notation we use throughout this thesis. We assume a certain basic knowledge of these topics. Thus, we do not introduced them in detail, but only roughly to make the notation clear. However, we also refer to further literature.*

## 2.1 Fundamentals

**Numbers** We use $\mathbb{Z}$, $\mathbb{N}$, $\mathbb{Q}$, and $\mathbb{R}$ to denote the set of integers and the sets of natural, rational, and real numbers, respectively. To restrict these sets we use self-explanatory subscripts, such as $\mathbb{R}_{\geq 0} = \{x \in \mathbb{R} : x \geq 0\}$. The natural numbers contain zero. To explicitly exclude zero we write $\mathbb{N}_{>0}$. If we have to extend the real numbers by infinity, we write $\mathbb{R} \cup \{\infty\}$. We define the minimum of the empty set to be infinity $\min \varnothing = \infty$.

**Sets and Subsets** Let $S$ be a set. By $|S|$ we denote its cardinality, by $2^S$ its power set, and by $\binom{S}{k}$ the set of all $k$-element subsets of $S$. Thus, it is $|2^S| = 2^{|S|}$ and $|\binom{S}{k}| = \binom{|S|}{k}$.

We denote the topological interior of a set $T \subseteq \mathbb{R}$ by

$$\operatorname{int}(T) = \{t \in T : (t - \varepsilon, t + \varepsilon) \subseteq T \text{ for some } \varepsilon > 0\}.$$

For $T_1, T_2 \subseteq \mathbb{R}$ we write

$$T_1 + T_2 = \{t_1 + t_2 : t_1 \in T_1, t_2 \in T_2\} \text{ and}$$
$$T_1 - T_2 = \{t_1 - t_2 : t_1 \in T_1, t_2 \in T_2\}.$$

If $T_2 = \{c\} \subseteq \mathbb{R}$, we also write $T_1 \pm c$ instead of $T_1 \pm T_2$.

**Functions** For a real-valued function $f \colon X \to \mathbb{R}$ and $\varepsilon \in \mathbb{R}$ we define $f + \varepsilon \colon X \to \mathbb{R}$ by $x \mapsto f(x) + \varepsilon$. Analogously, we define $f - \varepsilon$. In this thesis, we only deal with the case $X \subseteq \mathbb{R}$. If $f$ is piecewise linear, we denote its number of linear pieces by $\#\mathrm{p}(f)$.

## 2.2 Graphs and Networks

For detailed introductions to and rigorous definitions of graphs and networks we refer to the text books of Diestel [Die00], West [Wes00], as well as Ahuja, Magnanti, and Orlin [AMO93] although we deviate slightly from their notations.

**Directed Graphs** A *(directed) graph* $G$ consists of a finite non-empty *vertex set* $V(G)$, a finite *arc set* $A(G)$, and mappings that assign every arc $a \in A(G)$ one *start-vertex* $\alpha(a)$ and one *end-vertex* $\omega(a)$. However, we normally write $G = (V, A)$ with $V = V(G)$ and $A = A(G)$, and assume the mappings are given implicitly.

Two arcs are *parallel* if they have the same start-vertex and the same end-vertex. We represent an arc with start-vertex $u$ and end-vertex $v$ by $uv$ if there is no risk of confusion (for example, if the graph has no parallel arcs). For an arc $uv$ we call $u$ a *predecessor* of $v$ and $v$ a *successor* of $u$. For a vertex $v$ we denote the set of its predecessors by $N_G^{\text{in}}(v)$ and the set of its successors by $N_G^{\text{out}}(v)$. The sets of all arcs with end-vertex $v$ or with start-vertex $v$ are denoted by $\delta_G^{\text{in}}(v)$ and $\delta_G^{\text{out}}(v)$, respectively. *Indegree* $\deg_G^{\text{in}}(v) = |\delta_G^{\text{in}}(v)|$ and *outdegree* $\deg_G^{\text{out}}(v) = |\delta_G^{\text{out}}(v)|$ of $v$ are the corresponding cardinalities and the *degree* $\deg_G(v)$ of $v$ is the sum of in- and outdegree. If the graph is clear from the context, we will omit the subscript $G$. We write $\Delta(G)$ for the maximum degree, $\Delta^{\text{in}}(G)$ for the maximum indegree, and $\Delta^{\text{out}}(G)$ for the maximum outdegree of the graph $G$.

The *inverse graph* $G^{-1}$ has the same vertex and arc set as the graph $G$ but every arc reversed. That is, $G^{-1}$ has the arc $uv$ if and only if $G$ has the arc $vu$. Formally, this can be defined by exchanging the mappings $\alpha$ and $\omega$ of start- and end-vertices. For a subset $S \subseteq V(G)$ of the vertices of a graph $G$, the *induced subgraph* $G[S]$ has the vertex set $S$ and contains exactly the arcs $a \in A(G)$ that have both endpoints $\alpha(a), \omega(a) \in S$ in the restricted vertex set $S$.

**Undirected Graphs** An *undirected graph* $G$ is of a finite non-empty *vertex set* $V(G)$ with a finite *edge set* $E(G)$ and a mapping that assigns every edge $e \in E(G)$ two *end-vertices* $\gamma(e) \in \binom{V(G)}{2}$. As for directed graphs we usually write $G = (V, E)$ with $V = V(G)$ and $E = E(G)$, and assume the mapping $\gamma$ to be given implicitly. We also write $uv$ or $vu$ for an edge $e \in E$ with $\gamma(e) = \{u, v\}$. Note that this definition of undirected graphs prevents self-loops.

Within this thesis, almost all considered graphs are directed. The only exceptions are graphs for instances of decision problems, which we use in reductions.

**Paths** A *path* in a directed graph $G$ is a finite sequence $P = (v_0, a_1, v_1, \ldots, a_k, v_k)$ of vertices and arcs such that, for $1 \leq i \leq k$, the arc $a_i$ has start-vertex $\alpha(a_i) = v_{i-1}$ and end-vertex $\omega(a_i) = v_i$ [see *walk* in Wes00, Definition 1.2.2]. We call $v_0$ the *start-vertex* and $v_k$ the *end-vertex* of the path. A $u$-$v$-path is a path with start-vertex $u$ and end-vertex $v$. Since we abbreviate an arc $a$ from $u$ to $v$ by $uv$, it has no added value to include the arcs in the notation for a path. Thus, we usually describe a path only by its vertices $P = (v_0, \ldots, v_k)$.

By $P|_{uv}$ we denote the restriction of the path $P$ to the *subpath* between the two vertices $u$ and $v$, which we assume to be among $v_0, \ldots, v_k$. More precisely, we even assume that $u$ precedes $v$ on the path $P$. If the end-vertex $v_k$ of the path $P$ equals the start-vertex of a path $P'$, we can *concatenate* $P$ and $P'$ and write $P \circ P'$.

**Cuts** For two vertices $s$ and $t$ of a graph $G = (V, A)$, an *$s$-$t$-cut* is a partition $(S, T)$ of the vertex set $V$ into two sets such that $s \in S$ and $t \in T$. We denote by $\delta^{\text{out}}(S) \subseteq A$ all arcs with start-vertex in $S$ and end-vertex in $T$. The *cardinality* $|(S, T)|$ of the cut is defined as the amount $|\delta^{\text{out}}(S)|$ of arcs leaving $S$.

**Networks** We use the term *network* to denote a graph associated with further information, such as capacities on arcs or demands on vertices. Thus, all relevant information on an instance of a specific problem can be encoded in a network. For example, to represent an instance of the maximum flow problem as stated in [AMO93, Chapter 6], we would define a network consisting of a directed graph, arc capacities, and designated source and sink vertices.

## 2.3 Complexity Theory

The most important complexity classes for this work are **P** and **NP**. Roughly speaking, the class **P** contains all decision problems which are solvable in polynomial time by a deterministic algorithm. Similarly, the class **NP** contains those problems that are verifiable in polynomial time by a deterministic algorithm. A problem is ***NP**-hard* if it is at least as hard as every problem in **NP**. An **NP**-hard problem is ***NP**-complete* if it is also contained in **NP**.

Another complexity class, which is relevant for SFP in Chapter 6, is the class $\mathbf{\Sigma}_2^p$ from the second level of the polynomial hierarchy. One way to characterize this class is via oracle machines. Again roughly speaking, the class $\mathbf{\Sigma}_2^p$ contains those problems that are verifiable in polynomial time by a Turing machine that has access to an oracle for an **NP**-complete problem. This oracle essentially allows answering decision problems for the corresponding problem in constant time.

For an introduction to the classes **P** and **NP** with a comprehensive list of **NP**-complete problems we refer to the book of Garey and Johnson [GJ79]. For more detailed introductions and further insights into $\mathbf{\Sigma}_2^p$ we refer to Papadimitriou [Pap94, Section 17.2], Arora and Barak [AB09, Section 5.1], and Haan [Haa19, Section 2.2.1]. Of course, these books also contain chapters about the classes **P** and **NP**.

We always consider (polynomial-time) Karp reductions when proving the **NP**- or $\mathbf{\Sigma}_2^p$-hardness of problems. For more information on this aspect we again refer to Arora and Barak [AB09, Section 2.2].

If we write that a problem is *polynomially solvable* or *solvable in polynomial time*, it always means that it is solvable in a time that is polynomial in the input size.

Additionally, in Section 5.4 we prove the **APX**-hardness of MaxADP. Roughly speaking, the complexity class **APX** consists of those optimization problems (from the class **NPO**) that can be approximated up to a constant factor in polynomial time [see ACG+99, Definition 3.9]. This latter class **NPO** essentially contains the optimization problems corresponding to decision problem from **NP** [see ACG+99, Definition 1.17].

When proving **APX**-hardness for problems in **NPO** we use $L$-reductions as defined in [ACG+99, Definition 8.4], [Pap94, Section 13.2], or [PY91]. For more information about optimization problems, approximation algorithms, and approximation preserving reductions we refer to the textbook of Ausiello et al. [ACG+99] in general.

## 2.4 Linear and Integer Linear Programming

A few times in this thesis, we use linear and integer linear programs to formalize optimization problems. As this is a well-researched topic and lots of literature is available, the usage of these concepts allows us to easily obtain profound results. We distinguish *Linear Programs* (LP) that only contain fractional variables, *Integer Programs* (IP) that only contain integer variables, and *Mixed Integer Linear Programs* (MIP) that might contain fractional as well as integer variables. For an introduction to this topic we refer to Grötschel, Lovasz, and Schrijver [GLS88].

## 2.5 Multi-Criteria Optimization

Although the mathematical problems considered in Parts I and II of this thesis are exclusively single-criteria, the practical project is of an inherently multi-criteria nature. For its description in Chapter 3, we take up only the very basic concepts in multi-criteria optimization and refer to [Ehr05] for a detailed introduction to this topic.

Given a set $\mathcal{X}$ of feasible solutions and an objective $f\colon \mathcal{X} \to \mathbb{R}^p$ we are mostly interested in efficient solutions. Here, a solution $\hat{x} \in \mathcal{X}$ is *efficient* or *Pareto optimal* if no other solution has an objective vector that is at least as good in every and strictly better in at least one component. The image $f(\hat{x})$ of an efficient solution $\hat{x}$ is called *non-dominated.*

In our project, the feasible solutions correspond to paths in a graph. Since "efficient" is also used in common parlance, we refrain from writing "efficient paths" in the project description and instead use "non-dominated paths" in order to avoid confusion. Mathematically speaking, we identify the paths with their objective vector.

# Chapter 3

# Application Background

*In this chapter, we focus on our project with Deutsche Bahn, which motivated this thesis. We give an overview of the tasks and goals of this cooperation and work out the connection to the theoretical parts of this thesis. In short, the project was about the diversification of train paths offered to short-term train path requests in rail freight transport.*

## 3.1 Basic Concepts and Tasks

In essence, the project with DB Netze was about preparing offers for rail freight transport. Like Deutsche Bahn in Germany, railroad network operators offer industry customers the ability to reserve train paths (German *Trassen*) for freight trains. A customer request consists of route and train specifications. The route specification includes origin, destination, and potentially some intermediate points. At least one of them must be, but also several may be, provided with a desired departure or arrival time. The train specification contains information about the train, such as length, weight, maximum speed, or the type of locomotive. The task of the network operator is now to offer train paths for such a customer request.

A *train path* aggregates all the relevant spatial and temporal information about a possible route for the train. For every used track section, it contains exact entry and exit times. To meet the former requirements the course must lead from the origin via all intermediate points to the destination. Furthermore, the train path must approximately adhere to all specified departure and arrival times. To be compatible with the existing traffic, the train path may only use a track section if it is not already occupied in the required time interval. That also includes respecting a specific amount of buffer time (as trains may not run arbitrarily close) and track sections that lie behind break paths (these must be free in the event that the train does not stop in time).

Remember that we differentiate between *tracks* and *train paths*: by *tracks* we mean only the raw infrastructure whereas *train paths* add departure and arrival times to tracks.

The computation of train paths depends significantly on the representation of the rail network and the existing traffic. During the project, this representation changed fundamentally. The reason for this was to adapt to a change in the data structures used internally by DB Netze in order to be able to work with up-to-date data. We describe the data structures in the two project phases to Sections 3.4 and 3.5. In these sections, we also describe for each of the two cases how to address the computation of train paths and discuss challenges to be overcome.

In addition to the public transport timetable there is also a (long-term) working timetable for freight transport, where customers can apply for train paths to requests that are known in advance or that are required periodically. Furthermore, customers can order train paths at short notice. So far, these short-term requests have mostly been answered manually. The manual construction of these train paths has been technically supported for a long time but a fully automatic construction has only been in practical use for a few years. However, until now, this construction only works if the request meets certain conditions, and so far a customer is only offered a single train path.

The primary goal of this project was to investigate how customers could be offered various and especially different train paths. However, the project was also about the limited remaining capacities and how to use them optimally. These two aspects raise a number of questions, some of which are: How can different train paths be computed? What does *different* mean in this context? How should different train paths be evaluated and compared? Which paths should be offered to the customer? How do the individual train paths influence the remaining capacities, and how should this be measured?

The requirement that several, suitably different offers should be presented does not only occur in this setting. For example, car drivers also expect the navigation system to provide alternatives with less tolls or with a lower risk of traffic jams. These types of problems inspired us to consider the almost disjoint paths problem in Chapter 5.

## 3.2 Rating and Selecting Train Paths

We mainly focus on three parameters in order to evaluate train paths: distance, duration, and deviation. The *distance* is the length of the route, the *duration* is the total travel time including waiting, and the *deviation* is the sum of all deviations from the desired departure or arrival times specified by the customer. Moreover, we also include the utilization along the train paths in the selection process to avoid sections that are already heavily used. Additionally, we respect differences between the geographic courses of train paths as these play an important role in the subjective perception of difference. Thus, comparing and selecting train paths is an intrinsically multi-criteria problem.

**Selection Overview** First, we need a set of train paths from which to select. In this section, we assume such a set to be given and refer to Sections 3.4 and 3.5 for more information about computing such a set. Our selection process then works as follows. Initially, we rate the train paths by distributing points that measure how *good* a train path is. Hereafter, we successively choose the best-rated of the remaining train paths and adjust the scores of the rest. This adjustment is based on the geographic difference to the just selected train path. This procedure is repeated until we obtain the desired amount of paths.

**Initial Scores** The initial scores are determined based on scores for the aforementioned criteria, which we compute independently of each other. For distance, duration, and occupancy we put the corresponding value of every train path in relation to the best value among all train paths. That is, we divide, for example, the distance of every train path by the distance of the shortest path to obtain detour factors. This way we get a normalized value for every criterion. Only for the deviation we use absolute values as these are comparable, no matter how different the route specifications are. Based on these values we then compute points that penalize large discrepancies exponentially. The overall initial score of a train path is then composed of a weighted sum of the penalty points for the different criteria.

**Geographic Difference** The geographic difference between two train paths is determined by two measures. First, we look at the geographic courses and compute the area spanned between them. Again, this area is normalized based on the linear distance between origin and destination. Second, we use the railway facilities on both train paths. On the one hand, we compare how many facilities on the paths coincide and how many are contained in only one path. On the other hand, we also consider the smallest distances between these railway facilities.

## 3.3 Literature and Related Problems

In the course of climate change and the movement toward a more sustainable transport sector, rail freight transport has become an increasingly important topic. This triggers a lot of research in this area, especially since the rail infrastructure is limited and already heavily utilized today. We focus on the literature that is related to our situation.

Some literature specifically addresses the problem of scheduling additional trains into an existing timetable. However, the way infrastructure, existing traffic, track occupancies, and further requirements are modeled differs a lot. Also the level of detail and the degree of abstraction varies strongly. Finally, a large part of the literature formulates the problem as an integer linear program whereas we focus on network algorithms.

A model quite similar to our setting is described by Haehn, Ábrahám, and Nießen in [HÁN20]. They also investigate the task of scheduling freight trains under consideration of the existing traffic that is already scheduled on the network. Comparable to the problem in the second phase of our project they specify a train path by a *timed path*, which is a path in the underlying network where each vertex of the path is associated with departure and arrival times. A bit different to our setting is the underlying graph, which in [HÁN20] reflects the raw railway network in a very detailed way. This network is independent of both the currently considered train and the existing traffic, and it does not contain temporal information. It reminds of the topological network in our first phase (cf. Section 3.4) but the level of detail and the way they handle the time is closer to the snippet graphs in our second phase (cf. Section 3.5). The main difference to our time-dependent setting is that the duration of a path in [HÁN20] is the difference between the arrival and a fixed earliest starting time $t_{start}$. In contrast, our duration is the time span between the actual departure and the arrival. This small difference changes the problem fundamentally as we always have to keep track of the corresponding departure times. While we have to use functions as labels for fastest path algorithms in the second phase of the project, Haehn, Ábrahám, and Nießen can stick to single values. To compute fastest timed paths they adapt methods of a more theoretically focused paper [HP74] by Halpern and Priess to their setting. For more information, especially regarding [HP74], we also refer to the literature overview in Section 7.2.

Except from the algorithmic approach in [HÁN20] the further literature is predominantly based on mixed integer linear programming formulations.

The routing of freight trains with fixed existing traffic explicitly on the German railway network is considered in [BKS+16] and [Klu18]. The freight train routing problems defined therein assume a much coarser model compared to our situation. However, they look at the problem from a long-term, more strategic perspective and also schedule multiple trains at once. To this end, they build time-space networks and formulate the problem as a mixed integer linear program.

Cacchiani, Caprara, and Toth use similar methods in [CCT10] to schedule additional trains into an existing timetable. They also focus on freight trains and use a time-space network together with integer linear programming. As in [Klu18] they schedule multiple trains simultaneously. In contrast, they have a much more detailed model and thus consider the problem on a railway corridor rather than on the complete network.

Another work dealing with scheduling additional trains in existing timetables (related to the high-speed railway network in China) is [Tan14] by Tan. A major difference to our setting is that he allows modifying the given timetable which is excluded in our situation. However, Tan's PhD thesis covers different variants of the problems and contains a clear literature overview with helpful tables comparing problems with various characteristics. Tan assumes an initial cyclic timetable and mainly investigates mixed integer linear programs exploiting the periodic structure of constraints.

Mu and Dessouky also develop methods to schedule multiple freight trains in [MD11]. They use a detailed model and tackle the problem on large networks by heuristically decomposing it into smaller subproblems, which are then solved using mixed integer linear programs.

Weiß, Opitz, and Nachtigall describe an automated train path construction to schedule freight trains between already existing traffic in [WON14]. The approach is based on the software system TAKT [GWON12; Opi09] and uses the functionalities it provides.

Burdett and Kozan follow in [BK09] a heuristic approach to route multiple trains. They formulate the scheduling additional train services problem as a hybrid job shop scheduling problem in which trains correspond to jobs and track sections to machines. They schedule the trains one after the other in a greedy manner and make use of backtracking. To further improve the solution they apply a simulated annealing approach.

In the second phase of our problem we also take the utilization in the railway network into account. The goal is here to avoid heavily occupied track sections, which directs to [FGN09]. There, Flier, Graffagnino, and Nunkesser consider the scheduling of trains on corridors while minimizing the risk of delay. They use a time-expanded network in which they compute a set of train paths that are Pareto optimal with respect to travel time and risk. To determine the risk of a connection they use regression models based on real world delay data.

## 3.4 The First Phase: A Time-Expanded Network

We describe data, algorithms, and challenges in the first phase of the project. Here, the train path construction reduces to a shortest path problem in a time-expanded network whose sheer size is challenging. The need to avoid geographic cycles causes further difficulties as they necessitate resource constraints that make the shortest path problem much harder. In order to find various "good" paths, we compute all non-dominated solutions with respect to distance, duration, and deviation. However, before we describe our train path construction, we have to explain how train paths are represented and which data is given.

### Train Paths in the Time-Expanded Network

In the first phase of the project, the railway network is cut into many track segments that are equipped with several time corridors throughout the planning period. A time corridor is represented by a departure and an arrival time, and it is associated with minimum requirements on train specifications. Together with the underlying track segment we refer to such a time corridor as *train path segment*.

A train may only use a train path segment if it meets the required specifications. For example, it must be able to reach a certain speed in order to be able to achieve the desired arrival time at the end of the track section. Similar restrictions apply for other criteria such that a train also needs a minimum breaking power and can only have a certain length. Since a train path segment corresponds to a track segment, it has a specific length to which we refer as distance. With the associated departure and arrival times we can also assign a duration to every train path segment.

**Time-Expanded Network** All train path segments are connected to form a large time-expanded network. For two train path segments to fit together, arrival time and destination of the earlier one have to match with departure time and origin of the later one. However, further requirements have to be fulfilled as well, for example because the effect of switches depends on the passage direction. The arcs of the resulting network correspond to the train path segments and the vertices represent locations at certain points in time.

In particular, the time-expanded network is independent of the customer requests. The arcs represent track sections in time corridors during which these are still unoccupied. The structured arrangement in a graph locally models which train path segments we can connect. Usually, the planning period is one day with additional train path segments that start the previous day and others that last into the following day.

**Train Paths** A train path for a customer request corresponds to a path in the time-expanded network. Recall that a customer request consists of route information and train specifications. Concerning the route we are given an origin and a destination, potentially with desired departure and arrival times.

The train path has to start in a vertex that represents the origin at a time close to the desired departure. Accordingly, it has to end in a vertex of the destination at a time close to the desired arrival time. If intermediate points are specified, the train path has to respect these, too. Furthermore, only suitable arcs may be contained. That is, the train path may only use arcs for which the train specifications from the customer request fulfill the requirements of the respective train path segment.

We evaluate a train path, as already described in Section 3.2, with distance, duration, and deviation. In the first phase, we do not take the track utilization into account. The distance of the train path is the sum of the distances of the train path segments it consists of. The duration is the difference of the arrival time at the destination and the departure time at the origin. The deviation of a train path is calculated from the desired departure and arrival times. For every location that has a desired time we check when the train path passes through there and sum up the corresponding deviations.

## Computing Train Paths in the Time-Expanded Network

To compute a broad range of different train paths for a customer request, we determine all non-dominated paths. In order for this to be fast enough in practice, however, we have to restrict the time-expanded network to a sufficiently small subgraph. To speed up this restriction too, we need the topological network. In the following, we deal with these aspects and with the need to avoid geographic cycles.

**Multi-Criteria Shortest Path Algorithm** We determine all non-dominated paths with a multi-criteria label-correcting algorithm similar to [Ehr05, Algorithm 9.2]. At every vertex $v$ of the time-expanded network we store a label containing distance, duration, and deviation of every non-dominated path from a source vertex to the vertex $v$. To relax an arc $uv$ during the algorithm we add distance, duration, and deviation of the train path segment corresponding to the arc to the labels stored at $u$. We unite these with the labels already at $v$ and only keep the non-dominated ones.

**Topological Network** The vertices of the time-expanded network correspond to locations at certain points in time. Merging all vertices corresponding to identical locations results in the topological network. It solely represents spatial information and only contains a very limited amount of temporal information. Each of its arcs represents a set of arcs from the time-expanded network. We define distance (and duration) of arcs in the topological network as the minimum distance (duration) of represented arcs in the time-expanded network. Thus, shortest paths in the topological network (with respect to distance or duration) are lower bounds for paths between the corresponding locations in the time-expanded network. Due to the small size of the topological network, we can compute shortest path trees quickly.

**Selecting Subgraphs** We are only interested in "sufficiently good" solutions. A formalization of this vague statement on the part of Deutsche Bahn in this project phase is that we only want to offer paths whose distance (and duration) is at most twice as large as the distance (duration) of a shortest (fastest) path. Before we run the aforementioned multi-criteria shortest path algorithm, we restrict the time-expanded network to exactly the vertices that are contained in paths of appropriate distance and duration.

First, we restrict the time-expanded network with respect to the distances. That is, we compute the subgraph that consists of the vertices on paths that are at most twice as long as the distance of a shortest path. For this we use a multi-source A*-algorithm [HNR68] with the distances in the topological network as lower bounds. In a first run, we compute the distances from the source vertices until we know the distance $D$ of a shortest path in the time-expanded network. We continue with this run until a vertex for which the known distance from the source plus the heuristic distance to the target exceeds $2D$. That is, we stop this run when we first see a vertex that is guaranteed not to lie on a feasible path. Thereafter, in a similar second run, we use the A*-algorithm in the inverse graph to determine for every vertex on a feasible path the shortest distance to a target vertex. As spatial subgraph we can select all vertices for which the distances from a source vertex and to a target vertex sum up to at most $2D$.

Second, we can further restrict this spatial subgraph to all vertices that are also contained on paths that are fast enough. For this we use two runs of a multi-source Dijkstra algorithm [Dij59] (the second again in the inverse graph starting from the target vertices).

We start with the spatial subgraph because the topological network provides much better lower bounds for distances than for durations, which is presumably due to the inherent spatial structure of the topological network. The A*-algorithm allows us to translate the good lower bounds for distances into a fast selection of the spatial subgraph. For the further restriction we use Dijkstra's algorithm as the reduced search space does not compensate the overhead for the A*-algorithm in this situation.

**Avoiding Geographic Cycles** A requirement we have to cope with is that train paths must not contain geographic cycles. That is, we are not allowed to use two vertices of the time-expanded network that correspond to the same location. This requirement has less objective mathematical reasons than rather subjective ones. Driving a cycle feels like a detour although it might actually be the fastest or shortest path possible. Note that geographic cycles in a train path make sense in principle since it is not always possible to wait because this occupies track sections, which might be required by other trains.

We incorporate this restriction into our algorithms by introducing resources for the locations. This translates the shortest path problem into one with additional resource constraints, which makes the problem significantly harder. Note that in general the resource constrained shortest path problem is **NP**-complete [HZ80] whereas we can find a shortest path without resource constraints in polynomial time. In fact, introducing a resource constraint for every location would ensure that we could barely remove any labels in the multi-criteria label-correcting algorithm (a path can only dominate another if its used locations are a subset of those of the other). This would result in unacceptably large running times.

Instead, we first compute the non-dominated paths without any resource constraints. We check which locations are repeatedly contained in one of the non-dominated paths, add resource constraints for them, and run a resource constrained multi-criteria shortest path algorithm. We iterate this procedure until none of the non-dominated paths contains a geographic cycle anymore.

## 3.5 The Second Phase: A Time-Dependent Network

We describe data, algorithms, and challenges in the second phase of the project. During the construction of train paths we have to actively manage the temporal component. The flexibility in determining departure times opens up a lot of possibilities on the one hand, but complicates the train path construction enormously. To compute various paths, we concentrate on a weighted-sum approach.

### Train Paths in the Time-Dependent Network

In contrast to the setting in the first phase of the project (see Section 3.4) we now have an individual network for each customer request. The time-dependent model is much more detailed. This complicates the train path computation but opens up new capabilities in return. For example, this model allows us to respect the occupancy of track segments, which in turn makes it possible to better utilize remaining capacities.

**Snippet Graph** The relevant data for a customer request is bundled in a directed acyclic graph. In this graph, the vertices represent track segments and the arcs indicate which of them the train can use consecutively. In this phase, the track segments are associated with further information, too. We refer to a vertex representing such a segment together with all associated data as a *snippet* and therefore call the directed acyclic graph a *snippet graph*. The graph has one origin and one destination snippet.

Unlike in the first phase, however, the given temporal information associated with the snippets is relative and not absolute. For every snippet, a speed profile, specifically adapted to the customer request, determines when the train is where on the segment. To this end, the track segment is further divided into single infrastructure elements, for each of which the speed profile provides an interval $[t_0, t_1]$. A train starting this snippet at a time $t$ then occupies such an infrastructure element during the interval $[t + t_0, t + t_1]$.

The speed profile also provides a duration of a snippet. A train can only wait at the end of snippets at which this is permitted. If waiting is allowed, the snippet also provides the infrastructure elements that the train occupies while waiting there.

**Train Path** A train path in this setting is given by a path from the origin to the destination snippet on which every snippet is assigned a departure time. These departure times must fit to the time profiles of the snippets. That is, we may first depart at a snippet after we arrive at the predecessor. Furthermore, there may only be waiting time in between the arrival at the end of a snippet and the departure at the subsequent snippet if waiting is allowed there.

**Existing Traffic** Another restriction on the departure times is that they must not cause conflicts with already scheduled trains. For every train of the existing traffic we are given the infrastructure elements it uses, each associated with an absolute time interval during which it is occupied by the train. When specifying the (absolute) departure times for the snippets on the path in our snippet graph, we have to ensure that occupancies of infrastructure elements of our train always have a certain time gap to occupancies by already scheduled trains.

In Part II of this thesis, we simplify the problem of constructing train paths in this time-dependent setting into the Fastest Path with Time Profiles and Waiting (FPTPW) problem. Although we skip many requirements and only consider fastest paths without respecting distance or deviation, the resulting problem is still quite similar to the practical one described here.

## Computing Train Paths in the Time-Dependent Network

Unlike in the time-expanded setting we have to actively manage the time information while computing train paths. We restrict departure times to seconds (making them integers), but although the snippet graphs consist of relatively few vertices, very many possible time points have to be considered. To overcome these difficulties we first compute synchronized time profiles for the snippets and then run a shortest path algorithm that propagates functions as labels. However, it is far too cumbersome to compute all non-dominated train paths in this setting. Instead, we concentrate on the weighted-sum approach and compute shortest paths for various weights in order to obtain a diverse set of train paths. We now describe these aspects in more detail and finally also examine how we can respect the utilization in the network.

**Precomputing Time Profiles** Before we actually compute train paths, we determine for every snippet the time intervals during which we might depart, arrive, or wait. On the one hand, this is based on the desired departure and arrival times of the customer and on the other hand on the existing traffic. We use a maximum waiting time to bound these time intervals in case this is not implied by other track occupancies.

For a snippet we define *departure (arrival) time profiles* as the union of all time intervals during which we might depart (arrive). However, we not only determine these time profiles separately for all snippets but also synchronize them across the snippet graph. This way we ensure that for any point in a departure time profile there is a train path from the origin to the destination that departs exactly at this point. Accordingly, we reduce the search space of the much more complex train path computation as much as possible. For more details on the synchronization we refer to our theoretic considerations on this topic in Chapter 8.

**Computing Train Paths** As highlighted above we have to keep track of the temporal component while computing train paths. We do this by a shortest path algorithm that uses cost functions as labels. Such a function for a snippet $v$ maps every possible departure time $t$ to the minimum cost of a train path from the origin to $v$ that we can continue by departing at $v$ at time $t$. In Chapter 9 we define these functions for FPTPW and prove that they are piecewise linear. This remains true in the practical situation where we calculate the cost as a weighted sum of duration, distance, and deviation.

In the end, we are interested in a path corresponding to a minimum of the cost function at the destination. By maintaining predecessor information along with the cost functions, we can backtrack these to obtain such a path, see also Section 10.2.4.

To obtain the cost function at the destination we can compute all cost functions in the order of a topological sorting. However, this is not practical as the number of their linear pieces becomes quite large. Additionally, we collect a lot of information in this way, which we do not need for an optimum train path. In order to keep the search space as small as possible we use an A*-based algorithm that only propagates single linear pieces of the cost functions. Which piece to propagate is determined by its minimum cost value and an estimated cost to the destination. For these estimates we compute the shortest distance and duration from every snippet to the destination in the snippet graph by completely ignoring the temporal component. We will return to this in Section 10.3 at the end of Chapter 10, which covers fastest path algorithms for FPTPW from a mainly theoretical point of view.

**Respecting Utilization** The freedom to choose the exact departure times very flexibly allows respecting the utilization in the network. By knowing the existing traffic we can compute occupancy rates of a snippet depending on the exact departure time. Approximating these with piecewise linear functions allows us to integrate them in the train path computation. By choosing appropriate weights we can thus avoid heavily used track sections. Furthermore, we can use our information about already scheduled trains such that the train path nestles against the existing traffic as well as possible and, thus, try to preserve construction scopes for subsequently scheduled trains.

# Part I

# Almost Disjoint Paths and Forbidden Pairs

*In this part, we introduce the Almost Disjoint Paths (*ADP*) and the Separating by Forbidden Pairs (*SFP*) problem. The first,* ADP*, adapts the problem of finding $k$ arc-disjoint paths between vertices $s$ and $t$ by relaxing the disjointedness requirement: every two of the selected paths may share up to one arc. The latter,* SFP*, asks for $k$ arc pairs such that every $s$-$t$-path contains both arcs of at least one such pair.*

*The main results classify* ADP *to be* ***NP****-complete and* SFP *to be $\boldsymbol{\Sigma}_2^p$-complete. Furthermore, we show how to solve* ADP *for constant $k$ in polynomial time and analyze the relation between* ADP *and* SFP*. In particular, we prove that these problems form a weakly dual pair whose duality gap is unbounded in general.*

# Chapter 4

# Two Weakly Dual Problems

*We introduce the Almost Disjoint Paths (*ADP*) and the Separating by Forbidden Pairs (*SFP*) problem. Using integer programming formulations and their relaxations we prove that they are weakly dual and that their duality gap is unbounded in general. However, we also specify a class of graphs for which this duality gap disappears and on which the problems are thus even strongly dual.*

## Assumptions and Notes

Throughout this chapter, $G = (V, A)$ denotes a directed graph, $s, t \in V$ are two distinct source and target vertices, and $\mathcal{P}$ is the set of all $s$-$t$-paths.

Large parts of this chapter are joint work with Oliver Bachtler and Sven O. Krumke, which is already published in [BBK22].

## 4.1 The Problems ADP and SFP

In this section, we define the Almost Disjoint Paths (ADP) and the Separating by Forbidden Pairs (SFP) problem. After relating the latter to the Path Avoiding Forbidden Pairs (PAFP) problem, we conclude the section with non-restrictive assumptions, which facilitate the handling of both problems ADP and SFP.

### The Almost Disjoint Paths Problem

In many applications, customers receive various offers from which they can select one or between which they can switch. Usually, these offers should be as diverse as possible to provide the customer with many different options. This also applies to our practical project, see Section 3.1. In practice, however, there is often no precise specification of what *diverse* means and one has a lot of freedom in measuring it. In contrast, we now consider a fairly simple theoretical problem that reflects certain aspects of these application-oriented questions. To this end, we define the almost disjoint paths problem. An instance of this problem is given by a directed graph $G = (V, A)$ with designated source and target $s, t \in V$. Offers correspond to $s$-$t$-paths and we assume two paths to be sufficiently different if they have at most one arc in common. We formalize this in the following definition before we define the almost disjoint paths problem in Problem 4.2.

**Definition 4.1** (Almost Disjoint)**.** A set of paths in a graph is called *almost disjoint* if every two paths of this set have at most one arc in common.

**Problem 4.2** (ADP)**.** An instance of the *Almost Disjoint Paths* (ADP) problem is given by a directed graph $G = (V, A)$ with two vertices $s, t \in V$ and a natural number $k \in \mathbb{N}$. The question is, whether a set of $k$ almost disjoint $s$-$t$-paths exists.

**Problem 4.3** (MaxADP)**.** In the optimization variant MaxADP of ADP we are given a directed graph $G = (V, A)$ with two vertices $s, t \in V$. The goal is to find the maximum number of almost disjoint $s$-$t$-paths.

We address the problem ADP again in Chapter 5, where we focus on complexity theoretic results. In the course of this chapter, we regard the connection to its dual problem SFP, which we introduce in the following.

## The Separating by Forbidden Pairs Problem

In the separating by forbidden pairs problem we have to choose a set of arc pairs that cover all $s$-$t$-paths in a graph. As we did for ADP, we define a decision variant SFP as well as an optimization variant MinSFP for this problem.

**Problem 4.4** (SFP)**.** An instance of the *Separating by Forbidden Pairs* (SFP) problem is given by a directed graph $G = (V, A)$ with two vertices $s, t \in V$ and a natural number $k \in \mathbb{N}$. The question is, whether there exists a set of $k$ arc pairs, that is, $\mathcal{A} \subseteq \binom{A}{2}$ with $|\mathcal{A}| = k$, such that every $s$-$t$-path in $G$ contains both arcs of at least one chosen pair.

**Problem 4.5** (MinSFP)**.** In the optimization variant MinSFP of SFP we are given a directed graph $G = (V, A)$ and two vertices $s, t \in V$. The goal is to find a minimum set of arc pairs such that every $s$-$t$-path in $G$ contains both arcs of at least one chosen pair.

From a complexity-theoretic point of view we consider SFP in detail in Chapter 6. We expand on the weak duality to ADP in Section 4.2. Prior to that, we show that SFP adds another level on top of the well-known Path Avoiding Forbidden Pairs (PAFP) problem.

## Relation to the Path Avoiding Forbidden Pairs Problem

In order to relate our separating by forbidden pairs problem to the path avoiding forbidden pairs problem, we first give a definition of the latter one.

**Problem 4.6** (PAFP)**.** An instance of the *Path Avoiding Forbidden Pairs* (PAFP) problem is given by a directed graph $G = (V, A)$, two vertices $s, t \in V$, and a set $\mathcal{A} \subseteq \binom{A}{2}$ of arc pairs. The question is, whether an $s$-$t$-path exists that uses at most one arc of every pair in $\mathcal{A}$.

In this light, SFP asks for a set $\mathcal{A}$ of arc pairs such that the corresponding PAFP instance has no solution. By negating the Boolean question from PAFP, the separating by forbidden pairs problem basically precedes a universally quantified statement ("Every path uses both arcs of at least one pair in $\mathcal{A}$.") with an existentially quantified question ("Does a small set $\mathcal{A}$ with this property exist?"). Since the inner statement corresponds to the **NP**-complete [GMO76] PAFP problem, it seems reasonable to expect that SFP is $\mathbf{\Sigma}_2^p$-complete. Indeed, we prove this later in Theorem 6.8.

Instead of specifying forbidden pairs of arcs in a PAFP instance one can also consider forbidden pairs of vertices, which is actually common in the literature. However, these two variants are "essentially equivalent" as we point out in the following remark.

**Remark 4.7.** For most purposes, it is not that important whether we forbid pairs of arcs or pairs of vertices in a PAFP instance because we can convert both variants into each other in polynomial time by standard constructions.

To move from pairs of arcs to those of vertices we subdivide the relevant arcs: If an arc $uv$ is contained in a forbidden pair, we introduce a new vertex $w$ and replace $uv$ by the arcs $uw$ and $wv$. A forbidden pair containing $uv$ now contains the vertex $w$ instead.

For the converse, if pairs of vertices are forbidden, we split a vertex $v$ into two vertices $v^{\text{in}}$ and $v^{\text{out}}$, which we connect by an arc $v^{\text{in}}v^{\text{out}}$. We replace every arc $uv$ entering $v$ by the arc $uv^{\text{in}}$ and every arc $vw$ leaving $v$ by the arc $v^{\text{out}}w$. A forbidden pair containing the vertex $v$ now contains the arc $v^{\text{in}}v^{\text{out}}$ instead. $\triangleleft$

Note that arc pairs in the definitions of both SFP and PAFP each contain two different arcs. The following remark justifies this restriction.

**Remark 4.8.** If arc pairs may contain the same arc twice, we can make PAFP instances smaller and reduce SFP to the well-known problem of finding a minimum cut.

If a PAFP instance has a forbidden pair $p = \{a, a\}$ that consists of a single arc twice, two interpretations are possible: either the arc $a$ must not be contained in any path or the pair does not enforce any additional constraints. In the first case we can remove the arc $a$ and in the second case we can ignore the pair $p$.

Also, it does not make sense to consider pairs with twice the same arc in SFP. To see this, let $k$ denote the cardinality of a minimum $s$-$t$-cut. By Menger's Theorem we know that there are $k$ arc-disjoint $s$-$t$-paths [see Die00, Theorem 3.3.5]. Hence, we need at least $k$ pairs to separate $s$ and $t$. However, choosing a pair for each arc in the cut that contains this arc twice yields a feasible solution with $k$ pairs. Thus, the problem reduces to determining a minimum cut. $\triangleleft$

Finally, we note that one can exploit PAFP to model a restriction in the first phase of our practical project, which we describe in Section 3.4 in detail. In this project phase, the train paths correspond to paths in a time-expanded network that must not contain a geographic cycle. Partitioning the vertices according to their geographic location allows us to reformulate this requirement: every train path may contain at most one vertex from every block of the partition. Thus, for every location and every two vertices corresponding to this location we can introduce one forbidden pair. To obtain pairs of arcs instead of vertices we can alternatively pair the arcs entering vertices of the same location.

## Assumptions on ADP and SFP

We consider simplifying restrictions on instances for ADP and SFP that we may assume without loss of generality.

Our first assumption is based on the observation that arcs and vertices not contained in any $s$-$t$-path are also not contained in any forbidden pair of an optimal solution. Thus, they are irrelevant for ADP as well as for SFP and we may assume they do not exist.

**Assumption 4.9.** Every arc and every vertex is contained in an $s$-$t$-path.

Note that we can restrict the graph $G$ to a subgraph that fulfills Assumption 4.9 in linear time by two breadth-first or depth-first searches: the graph fulfilling Assumption 4.9 is the subgraph induced by all vertices reachable from $s$ in $G$ and reachable from $t$ in $G^{-1}$.

Our second assumption concerns the direct arc $st$, which itself forms an $s$-$t$-path of length one. Since the two arcs in a forbidden pair of an SFP instance are distinct, see Remark 4.8, this path never contains both arcs of a forbidden pair. Consequently, if $G$ contains the direct arc $st$, we cannot separate $s$ and $t$ by forbidden pairs, no matter how large the number $k$ of forbidden pairs is. Regarding ADP we can add the path formed by the arc $st$ to every set of almost disjoint paths while maintaining this property: a path consisting of a single arc can have at most one arc with any other path in common. To exclude this case we assume that this arc does not exist.

**Assumption 4.10.** The graph $G$ does not contain the arc $st$.

## 4.2 Weak Duality

In this section, we show that ADP and SFP are weakly dual by providing integer programming formulations with suitable linear relaxations that form a dual pair. Thereafter, we construct graphs on which the duality gap between ADP and SFP becomes arbitrarily large and close the section with showing that the duality gap disappears on every graph that contains an $s$-$t$-cut with a single outgoing arc.

### Exponential-Size IP Formulations

We provide integer programming formulations for MAXADP and MINSFP based on the set $\mathcal{P}$ of all $s$-$t$-paths and on the set $\binom{A}{2}$ of all possible arc pairs.

For MAXADP we introduce for every path $P \in \mathcal{P}$ a binary variable $y_P$ that indicates whether we choose this path or not. This allows us to formulate MAXADP as the following integer program.

$$\begin{aligned}
\max \quad & \sum_{P \in \mathcal{P}} y_P && && \text{(4.1a)}\\
\text{s.t.} \quad & \sum_{P \in \mathcal{P} : \{a_1, a_2\} \subseteq A(P)} y_P \le 1 && \forall \{a_1, a_2\} \in \binom{A}{2} && \text{(4.1b)}\\
& y_P \in \{0, 1\} && \forall P \in \mathcal{P} && \text{(4.1c)}
\end{aligned}$$

Similarly, we formulate MINSFP by introducing a binary variable $x_{\{a_1,a_2\}}$ for every pair of arcs $\{a_1, a_2\} \in \binom{A}{2}$. Such a variable indicates whether we choose the corresponding pair as a forbidden one.

$$\begin{aligned}
\min \quad & \sum_{\{a_1,a_2\} \in \binom{A}{2}} x_{\{a_1,a_2\}} && && \text{(4.2a)}\\
\text{s.t.} \quad & \sum_{\{a_1,a_2\} \in \binom{A(P)}{2}} x_{\{a_1,a_2\}} \ge 1 && \forall P \in \mathcal{P} && \text{(4.2b)}\\
& x_{\{a_1,a_2\}} \in \{0, 1\} && \forall \{a_1, a_2\} \in \binom{A}{2} && \text{(4.2c)}
\end{aligned}$$

Note that the number of $s$-$t$-paths in $G$ might be exponential in the size of $G$. Accordingly, both Programs (4.1) and (4.2) are in general exponentially large.

We obtain straightforward linear relaxations of the Programs (4.1) and (4.2) by replacing the integrality constraints $y_P \in \{0, 1\}$ and $x_{\{a_1,a_2\}} \in \{0, 1\}$ by non-negativity constraints $y_P \ge 0$ and $x_{\{a_1,a_2\}} \ge 0$, respectively.

For MaxADP we obtain Program (4.3) as a linear relaxation of Program (4.1).

$$\begin{aligned} \max \quad & \sum_{P \in \mathcal{P}} y_P && && \text{(4.3a)}\\ \text{s.t.} \quad & \sum_{P \in \mathcal{P}: \{a_1,a_2\} \subseteq A(P)} y_P \leq 1 && \forall \{a_1,a_2\} \in \binom{A}{2} && \text{(4.3b)}\\ & y_P \geq 0 && \forall P \in \mathcal{P} && \text{(4.3c)} \end{aligned}$$

For MinSFP we obtain Program (4.4) as a linear relaxation of Program (4.2).

$$\begin{aligned} \min \quad & \sum_{\{a_1,a_2\} \in \binom{A}{2}} x_{\{a_1,a_2\}} && && \text{(4.4a)}\\ \text{s.t.} \quad & \sum_{\{a_1,a_2\} \in \binom{A(P)}{2}} x_{\{a_1,a_2\}} \geq 1 && \forall P \in \mathcal{P} && \text{(4.4b)}\\ & x_{\{a_1,a_2\}} \geq 0 && \forall \{a_1,a_2\} \in \binom{A}{2} && \text{(4.4c)} \end{aligned}$$

We denote Programs (4.1) and (4.2) by $\mathrm{IP}^{\mathrm{ADP}}$ and $\mathrm{IP}^{\mathrm{SFP}}$, respectively. Accordingly, let $\mathrm{LP}^{\mathrm{ADP}}$ and $\mathrm{LP}^{\mathrm{SFP}}$ denote the corresponding linear relaxations given by Programs (4.3) and (4.4). Writing $c(\Pi)$ for the optimal objective value of a program $\Pi$, we obtain

$$c(\mathrm{IP}^{\mathrm{SFP}}) \geq c(\mathrm{LP}^{\mathrm{SFP}}) = c(\mathrm{LP}^{\mathrm{ADP}}) \geq c(\mathrm{IP}^{\mathrm{ADP}}). \tag{4.5}$$

The two inequalities hold since $\mathrm{LP}^{\mathrm{ADP}}$ and $\mathrm{LP}^{\mathrm{SFP}}$ are relaxations of $\mathrm{IP}^{\mathrm{ADP}}$ and $\mathrm{IP}^{\mathrm{SFP}}$, respectively. The equality is due the (strong) duality of linear programming [see, for example, MG07, Section 6.1] since Program (4.4) is the dual of Program (4.3) and vice versa. Hence, ADP and SFP form a pair of dual problems.

A direct consequence of this duality is that the size of a set of almost disjoint $s$-$t$-paths is a lower bound on the number of forbidden pairs required to separate $s$ and $t$. This is also intuitively clear since no pair of arcs can be contained in two paths that only have a single arc in common. Thus, a separating set of forbidden pairs has to contain at least one pair for each path of a set of almost disjoint paths.

## Duality Gap

In the following, we show that ADP and SFP are only weakly dual in general. Therefore, we consider their *duality gap* $c(\mathrm{IP}^{\mathrm{SFP}}) - c(\mathrm{IP}^{\mathrm{ADP}})$. By Equation (4.5) it is the sum of the *integrality gap* $c(\mathrm{LP}^{\mathrm{ADP}}) - c(\mathrm{IP}^{\mathrm{ADP}})$ between the MaxADP formulations from Programs (4.1) and (4.3) and of the *integrality gap* $c(\mathrm{IP}^{\mathrm{SFP}}) - c(\mathrm{LP}^{\mathrm{SFP}})$ between the MinSFP formulations from Programs (4.2) and (4.4).

In Example 4.11 we provide an instance with a positive integrality gap between $\mathrm{IP}^{\mathrm{ADP}}$ and $\mathrm{LP}^{\mathrm{ADP}}$. Afterwards, in the proof of Lemma 4.12, we extend this example to a family of instances for which this integrality gap is unbounded.

**Example 4.11.** An instance with $4 = c(\mathrm{IP}^{\mathrm{SFP}}) = c(\mathrm{LP}^{\mathrm{ADP}}) > c(\mathrm{IP}^{\mathrm{ADP}}) = 2$ is given in Figure 4.1. It consists of a path of length four where every arc is doubled.

We first show $c(\mathrm{IP}^{\mathrm{ADP}}) = 2$. A maximum set of almost disjoint paths contains at least one path. Without loss of generality we assume that this path uses only upper arcs (by swapping upper and lower arcs). Thus, every other path in this set has to use at least three lower arcs. But two paths using three lower arcs have at least two lower arcs in common. Since there are two disjoint paths, we have $c(\mathrm{IP}^{\mathrm{ADP}}) = 2$.

We continue with $c(\mathrm{IP}^{\mathrm{SFP}}) = 4$. It holds $|\mathcal{P}| = 2^4 = 16$ and for a pair $\{a_1, a_2\} \in \binom{A}{2}$ of non-parallel arcs we have $|\{P \in \mathcal{P} : \{a_1, a_2\} \subseteq A(P)\}| = 2^2 = 4$. Note that choosing a pair of parallel arcs for SFP does not make sense as no $s$-$t$-path uses both of these. Hence, $y \equiv 1/4$ is a feasible solution for $\mathrm{LP}^{\mathrm{ADP}}$ with an objective value of 4. Thus, we have $c(\mathrm{IP}^{\mathrm{SFP}}) \geq c(\mathrm{LP}^{\mathrm{ADP}}) \geq 4$ and since four forbidden pairs are also sufficient to separate $s$ and $t$, we obtain $c(\mathrm{IP}^{\mathrm{SFP}}) = c(\mathrm{LP}^{\mathrm{ADP}}) = 4$. $\triangleleft$

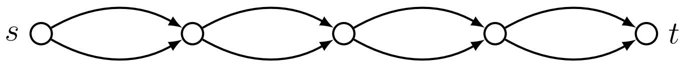


Figure 4.1: A graph for Example 4.11 with an integrality gap for ADP.

**Lemma 4.12.** The integrality gap $c(\mathrm{LP}^{\mathrm{ADP}}) - c(\mathrm{IP}^{\mathrm{ADP}})$ for ADP is unbounded. Moreover, also the ratio $c(\mathrm{LP}^{\mathrm{ADP}})/c(\mathrm{IP}^{\mathrm{ADP}})$ can become arbitrarily large.

*Proof.* We extend the graph from Figure 4.1 to a family of instances. For this purpose, let $P_\ell^k$ denote an $s$-$t$-path of length $\ell$ whose $\ell$ arcs are all replaced by bunches of $k$ parallel arcs. Assume $k \geq 2$ and $\ell \geq k + 1 + k(k-1)/2$. We prove $c(\mathrm{LP}^{\mathrm{ADP}}) = k^2 > k = c(\mathrm{IP}^{\mathrm{ADP}})$.

We first observe that $k^2$ pairs are sufficient to separate $s$ and $t$: select two bunches of parallels and choose all pairs containing exactly one arc from each of these bunches. Hence, $k^2 \geq c(\mathrm{IP}^{\mathrm{SFP}}) \geq c(\mathrm{LP}^{\mathrm{SFP}}) \geq c(\mathrm{LP}^{\mathrm{ADP}})$.

The graph $P_\ell^k$ has $k^\ell$ different $s$-$t$-paths. For a fixed pair of arcs we either have 0 or $k^{\ell-2}$ many $s$-$t$-paths using this pair (depending on whether both arcs are contained in the same bunch or not). Therefore, $y \equiv 1/(k^{\ell-2})$ is feasible for $\mathrm{LP}^{\mathrm{ADP}}$ and has an objective value of $k^\ell \cdot 1/(k^{\ell-2}) = k^2$. Using Equation (4.5) we obtain

$$k^2 \leq c(\mathrm{LP}^{\mathrm{ADP}}) = c(\mathrm{LP}^{\mathrm{SFP}}) \leq c(\mathrm{IP}^{\mathrm{SFP}}) \leq k^2$$

and, thus, $c(\mathrm{LP}^{\mathrm{ADP}}) = k^2$.

We now show that $P_\ell^k$ admits at most $k$ almost disjoint paths proving $c(\mathrm{IP}^{\mathrm{ADP}}) = k$. To this end, let $\mathcal{Q}$ be a maximum set of almost disjoint $s$-$t$-paths and enumerate these paths arbitrarily. Since the graph has $k$ disjoint $s$-$t$-paths, we have $|\mathcal{Q}| \geq k$. The second path has at most one arc in common with the first path and, if this is the case, we can assume without loss of generality that it is contained in the first bunch. More general, the $i$-th path has at most one arc in common with any of the first $i-1$ paths and we can assume (again without loss of generality) that these arcs are contained in the first $\sum_{j=1}^{i-1} j$ bunches.

Hence, if two of the first $k$ paths in $\mathcal{Q}$ have an arc in common, we can assume that it is in the first $\sum_{j=1}^{k-1} j = k(k-1)/2$ bunches. Our assumption $\ell \geq k+1+k(k-1)/2$ thus implies that each arc from the last $k+1$ bunches is contained in at most one of the first $k$ paths. Each bunch consists of $k$ arcs. Thus, each arc of the last $k+1$ bunches is contained in exactly one of the first $k$ paths. Any further path in $\mathcal{Q}$ has to use one arc from each of the last $k+1$ bunches. However, this means that it has at least two arcs in common with one of the first $k$ paths. As a consequence, there is no further path and it follows $c(\mathrm{IP}^{\mathrm{ADP}}) = |\mathcal{Q}| = k$. □

In particular, Lemma 4.12 together with Equation (4.5) directly implies that the duality gap between ADP and SFP is unbounded, which we state in the following corollary.

**Corollary 4.13.** The duality gap between ADP and SFP is unbounded.

In contrast to Lemma 4.12 we are not aware of any instance with an integrality gap for SFP. In this light, the formulation of MinSFP given by Program (4.2) seems to be quite strong. This fits together with the following observation that, if $\mathbf{P} \neq \mathbf{NP}$, the linear relaxation (4.4) cannot be solved in polynomial time.

**Lemma 4.14.** Program (4.4) cannot be solved in polynomial time unless $\mathbf{P} = \mathbf{NP}$.

*Proof.* Because the separation problem for the integral program (4.2) is the $\mathbf{NP}$-complete PAFP problem, also the separation problem for the relaxation (4.4) is $\mathbf{NP}$-hard. By [CCPS98, Theorem 6.36], the claim follows if we can prove that the polyhedra defined by Constraint (4.4b) form a proper class. To this end, we have to prove that, given a directed acyclic graph $G = (V, A)$, the dimension of each polyhedron and the maximum encoding size of Constraint (4.4b) for a path $P \in \mathcal{P}$ can be computed in polynomial time.

The dimension of the polyhedron corresponding to the graph $G$ is $\binom{|A|}{2} \in \mathcal{O}\left(|A|^2\right)$ which can be computed in polynomial time. And as the coefficients in Constraint (4.4b) are all either 0 or 1, we can also compute the maximum encoding length of such an inequality in polynomial time. □

We conjecture that the polyhedra of the linear programming relaxation (4.4) are in fact integral, or that at least all "relevant" extreme points are integral. However, the constraint matrix of Program (4.4) is not totally unimodular as the following example shows.

**Example 4.15.** For the SFP instance depicted in Figure 4.2, the constraint matrix of Program (4.4) is not totally unimodular: the sub-matrix corresponding to the pairs $\{sv_1, v_1v_2\}$, $\{v_2v_3, v_3v_4\}$, $\{v_4v_5, v_5t\}$ and the three paths that each use exactly one bend arc has determinant

$$\det \begin{pmatrix} 0 & 1 & 1 \\ 1 & 0 & 1 \\ 1 & 1 & 0 \end{pmatrix} = 2. \qquad \triangleleft$$

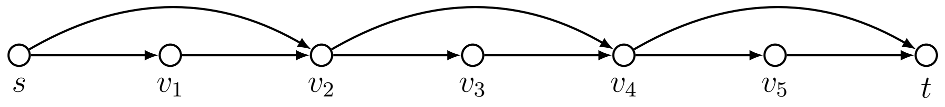


Figure 4.2: An SFP instance for which the constraint matrix is not totally unimodular. See Example 4.15 for more information.

Although the duality gap between ADP and SFP is in general unbounded, this is not always the case. For example, if we restrict ourselves to graphs that have an $s$-$t$-cut with a single outgoing arc, the duality gap disappears. In particular, for these instances neither the MaxADP nor the MinSFP formulation has an integrality gap.

**Lemma 4.16.** If $G$ has an $s$-$t$-cut $(S, T)$ with a single outgoing arc $\delta^{\text{out}}(S) = \{uv\}$, the duality gap is zero and we can solve ADP and SFP in polynomial time.

*Proof.* Note that every $s$-$t$-path in $G$ must use the arc $uv$, so the paths in a set of almost disjoint $s$-$t$-paths must be disjoint aside from $uv$. We can compute maximum sets $\mathcal{P}_s$ and $\mathcal{P}_t$ of disjoint $s$-$u$- and $v$-$t$-paths, respectively. Combining $k = \min\{|\mathcal{P}_s|, |\mathcal{P}_t|\}$ paths from each of these sets results in $k$ almost disjoint $s$-$t$-paths. In addition, one of the subgraphs $G[S]$ or $G[T]$ has a cut with $k$ outgoing arcs. Bundling $uv$ with each of these arcs results in $k$ forbidden pairs separating $s$ and $t$. $\square$

We will encounter this type of graphs again in Section 6.1 when we analyze a heuristic for MinSFP. It is stated in Algorithm 6.1 on Page 54 and extends the idea from the proof of Lemma 4.16.

## 4.3 Literature

As already mentioned at the start of Section 4.1, there are many applications, in which customers receive multiple offers that should be as diverse as possible. A common use case is the construction of alternative routes in transportation or road networks. These make sense in this context, for example to avoid route closures, heavy traffic, or tolls. Another example where alternative routes are of use is to distribute risk. For example, if dangerous goods need to be transported regularly, alternative routes that affect different people allow for an equal risk distribution amongst the people exposed. Several practical algorithms for computing alternative routes have been developed [see, for example, ADGW10; AEB00; BDGS11; DGS05; JKPK09].

On the graph-theoretic side, the (arc- or vertex-) disjoint paths problem is well-studied. Determining a maximum amount of disjoint $s$-$t$-paths can easily be done using maximum flow techniques [AMO93]. By Menger's theorem [Men27; Die00] this amount is equal to the minimum number of arcs needed to separate $s$ from $t$. This result follows from the max-flow min-cut theorem [DF57], which shows that these two problems form a strongly dual pair.

The following extension of the disjoint paths problem is also well-understood: given $k$ pairs of terminals $(s_1, t_1), \dots, (s_k, t_k)$, the objective is to find disjoint $s_i$-$t_i$-paths. For undirected graphs it is solvable in polynomial time if $k$ is constant (see [RS95] for a cubic and [KKR12] for a quadratic algorithm) and **NP**-complete in general [EIS76]. In the case of directed graphs, a single path is easy and two paths are already **NP**-complete [FHW80]. The problem remains **NP**-complete for few paths even on very restricted graph classes like acyclic, Eulerian, or planar graphs [Vyg95].

Another possible extension is to ask for $k$ disjoint $s$-$t$-paths that are short, which again makes sense for routing purposes. Suurballe describes an algorithm for this problem that is based on shortest path labelings [Suu74]. It is possible to combine both extensions and ask for shortest paths between different terminals. Eilam-Tzoreff shows that these problems in all configurations (for directed and undirected graphs with vertex- or arc-disjoint paths) are **NP**-complete and also provides a polynomial algorithm for two paths in an undirected graph with positive edge-weights [Eil98]. Berczi and Kobayashi present a polynomial algorithm for the directed version, also with two paths and positive arc-lengths, in [BK17].

In contrast, the same problem where the paths need not be completely disjoint has not garnered as much attention in the literature. The natural choice of allowing paths to have at most one arc in common leads to the Almost Disjoint Paths (ADP) problem that we defined in Problem 4.2 on Page 26.

Most of the literature on nearly disjoint paths is of a very practical nature as is evidenced by the initial examples we presented. We now discuss some of the (rarer) theoretical results that exist for problems similar to ADP. In [LJYZ18], Liu et al. introduce the $k$

shortest paths with diversity problem, in which the goal is to find a set of sufficiently dissimilar paths of maximum size (bounded by $k$). Of such sets, the one that minimizes the total path length is optimal. For this problem, an (incorrect) **NP**-hardness proof as well as a greedy framework is presented. In [CBG+18], Chondrogiannis et al. rectify said **NP**-hardness proof, showing that the problem is indeed strongly **NP**-hard, and develop an exact algorithm for it as well as heuristics. Moreover, Chondrogiannis et al. consider the problem of finding $k$ shortest paths with limited overlap [CBG+20]. This differs from the previous problem by requiring the $k$ paths to exist (instead of looking for a maximal set of up to $k$ paths). They prove that this variant is weakly **NP**-hard and develop two exact algorithms for it, one of which is pseudo-polynomial if $k$ is constant. The problems here are similar to ADP in the sense that they look for paths that are sufficiently dissimilar, though the measures used always result in similarity values between 0 and 1 because they compare the amount of arcs in common with some function based on the lengths of the two paths. Additionally, they want to minimize the total length of the paths found.

Inspired by the strong duality of max-flow and min-cut, we make analogous considerations for our almost disjoint paths problem and its dual, the Separating by Forbidden Pairs (SFP) problem. While the linear programming relaxations (4.3) and (4.4) of ADP and SFP form a dual pair and thus have the same objective value [GKT51], the corresponding integer versions (4.1) and (4.2) are only weakly dual. Note that in the min-cut problem we select an arc on every $s$-$t$-path whereas in SFP we select a pair of arcs on every $s$-$t$-path.

Apart from being dual to ADP, the separating by forbidden pairs problem adds another level on top of the well-known Path Avoiding Forbidden Pairs (PAFP) problem, see Section 4.1 for further information. Originating from the field of automated software testing [KSG73], PAFP also has applications in aircraft routing [BBB+15] and biology, for example in peptide sequencing [CKT+00] or predicting gene structures [KVB09]. The PAFP is **NP**-complete [GMO76] and various restrictions on the set of forbidden pairs have been considered. The problem becomes solvable if the pairs satisfy certain symmetry properties [Yin97] or if they have a hierarchical structure [KP09] while it remains **NP**-hard even if the pairs have a halving structure [KP09] or no two pairs are nested [Kov13]. The structure of the PAFP polytope has been analyzed [BBB+15] and Hajiaghayi et al. show that determining a path that uses a minimal amount of forbidden pairs cannot have a sublinear approximation algorithm [HKKM10]. Furthermore, forbidden pairs were also transferred to other combinatorial problems, for example minimum spanning trees [KLM13], matchings [AJKS20], and regular languages [CL16].

Note that some of the referenced papers consider forbidden pairs of vertices instead of pairs of arcs. However, these two variants can be converted into one another by standard constructions as we have shown in Remark 4.7. Moreover, as this problem is usually regarded on acyclic graphs, we also specifically regard ADP and SFP under this restriction.

## Conclusion

We defined the almost disjoint paths and the separating by forbidden pairs problem. With exponential-size integer programming formulations we proved that they form a dual pair and constructed a class of examples on which their duality gap is unbounded. Thus, in contrast to the max-flow and min-cut problems, ADP and SFP are only weakly dual. However, on graphs that have an $s$-$t$-cut with a single outgoing arc, the duality gap disappears and we can solve both problems in polynomial time.

This, however, is in general not the case. As we have seen in Section 4.1, SFP adds another level on top of the path avoiding forbidden pairs problem. The **NP**-completeness of the latter already suggests that SFP is $\mathbf{\Sigma}_2^p$-complete. Before we prove this in Chapter 6, we first prove the **NP**-completeness of ADP in the following chapter.

# Chapter 5

# Almost Disjoint Paths

*We study the complexity of the Almost Disjoint Paths (*ADP*) problem and prove that it is* ***NP****-complete and that* MaxADP *is* ***APX****-hard. Moreover, we provide a dynamic program that allows solving* ADP *for constant $k$ in polynomial time.*

## Assumptions and Notes

Throughout this chapter, $G = (V, A)$ denotes a directed graph and $s, t \in V$ are distinct source and target vertices in this graph. The following assumptions apply.

**Only Relevant Arcs and Vertices** → see Assumption 4.9 on Page 28
Every arc and every vertex is contained in an $s$-$t$-path.

**No Direct Arc** → see Assumption 4.10 on Page 28
The graph $G$ does not contain the arc $st$.

We formally defined ADP in Problem 4.2 on Page 26. Recall that it asks whether $G = (V, A)$ contains a set of $k$ almost disjoint $s$-$t$-paths.

Large parts of this chapter are joint work with Oliver Bachtler and Sven O. Krumke, which is already published in [BBK22].

## 5.1 Constantly Many Paths

We first show how to solve ADP for $k \leq 2$ before we present a dynamic program that solves the problem for any constant $k$ in polynomial time.

For $k = 1$, ADP reduces to reachability, which can be solved in linear time, for example by a breadth- or depth-first search. We can check whether two almost disjoint paths exist by computing one maximum flow per arc. For $a \in A$ we can define arc capacities $c_a$ with $c_a(a) = 2$ and $c_a(a') = 1$ for $a' \in A \setminus \{a\}$. An $s$-$t$-flow with respect to $c_a$ of value $\ell$ corresponds to $\ell$ many $s$-$t$-paths that have at most the arc $a$ in common. Therefore, two almost disjoint paths exist if and only if an $s$-$t$-flow of value at least 2 exists with respect to arc capacities $c_a$ for some arc $a$. Hence, we can solve the problem for $k = 2$ by computing a maximum $s$-$t$-flow with respect to $c_a$ for each $a \in A$ and checking whether one of them has value at least 2. In fact, instead of computing a maximum flow for each capacity $c_a$ it suffices to make (at most) two flow augmentations, which only require linear time. We obtain the following result.

**Lemma 5.1.** For $k = 2$, ADP can be solved in $\mathcal{O}(|A|(|V| + |A|))$ time.

However, this technique does not generalize to $k > 2$ because in this case several arcs might be contained in multiple paths and we cannot guarantee that two paths only share a single arc. Instead we use a dynamic program to find a constant number $k$ of almost disjoint $s$-$t$-paths in polynomial time.

**Theorem 5.2.** For constant $k$, ADP is polynomial time solvable on acyclic graphs.

*Proof.* Let $m = |A|$ be the number of arcs and assume $k$ to be fixed. By assumption $G$ is acyclic and therefore admits a topological ordering $\nu\colon V \to \mathbb{N}$ of its vertices. Assumption 4.9 implies that the source $s$ (the target $t$) always has the smallest (largest) value of $\nu$. We now describe a dynamic program solving ADP.

**States** The dynamic program is based on states. A state $((a_1, \dots, a_k), \mathcal{I})$ consists of $k$ (not necessarily disjoint) arcs and an intersection pattern $\mathcal{I} \subseteq \binom{\{1,\dots,k\}}{2}$. We associate a state with a Boolean value $x((a_1, \dots, a_k), \mathcal{I})$ that is `true` if and only if $k$ almost disjoint paths $P_1, \dots, P_k$ with the following properties exist:

- ▷ For every $i \in \{1, \dots, k\}$ the path $P_i$ is an $s$-$\omega(a_i)$-path whose last arc is $a_i$.
- ▷ For $i \neq j$ the paths $P_i$ and $P_j$ have an arc in common if and only if $\{i, j\} \in \mathcal{I}$.

There are $m^k$ different possibilities to choose $k$ out of $m$ arcs (with replacement). Additionally, we have $\mathcal{O}(2^{k^2})$ different intersection patterns, yielding $\mathcal{O}(m^k 2^{k^2})$ states in total. Note that this amount is polynomial since we assume $k$ to be constant.

**Comparing States** To enable the computation of the truth values of all states with a dynamic program we have to order them appropriately. For this purpose, we introduce the relation $\prec$ on the states. Using the topological ordering $\nu$ we define that $((a_1, \dots, a_k), \mathcal{I}) \prec ((a'_1, \dots, a'_k), \mathcal{I}')$ applies if and only if

$$\begin{aligned} &\nu(\alpha(a_i)) \leq \nu(\alpha(a'_i)) \text{ for all } i \in \{1, \dots, k\} \text{ and} \\ &\nu(\alpha(a_i)) < \nu(\alpha(a'_i)) \text{ for at least one } i \in \{1, \dots, k\}. \end{aligned}$$

That is, we ignore the intersection pattern and compare the values in the topological ordering of the arc's start-vertices for each of the $k$ components separately.

**Goal** If we know the truth values of all states, we can determine whether $k$ almost disjoint $s$-$t$-paths in $G$ exist. We only have to check whether a state with value `true` exists whose arcs all enter the target $t$. In fact, we can check this during the dynamic program when computing the truth values of the appropriate states.

**Base** Similarly, we proceed at the start by determining the truth values of all states whose arcs all leave the source $s$. For any $k$ arcs $a_1, \dots, a_k \in \delta^{\text{out}}(s)$ and an arbitrary intersection pattern $\mathcal{I}$ we have that the value $x((a_1, \dots, a_k), \mathcal{I})$ is `true` if and only if $\mathcal{I} = \{\{i, j\} : a_i = a_j \text{ for } i, j \in \{1, \dots, k\}, i \neq j\}$. Note that these states actually suffice as base due to Assumption 4.9, which states that all arcs are contained in $s$-$t$-paths.

**Recursion** The dynamic program is based on a recursion that allows the computation of the truth value of a state based on the truth values of smaller states (with respect to $\prec$). To derive this recursion let $((a_1, \dots, a_k), \mathcal{I})$ be a state. If all arcs $a_1, \dots, a_k$ start in $s$, we are in the base case, which is already handled in the previous paragraph. Otherwise, an arc $a \in \{a_1, \dots, a_k\}$ maximizing the value $\nu(\alpha(a))$ satisfies $\alpha(a) \neq s$. Without loss of generality we assume $a = a_1$.

If $a_1 = a_i$ but $\{1, i\} \notin \mathcal{I}$ for some $i \in \{2, \dots, k\}$, the state must have truth value `false` as any paths $P_1$ and $P_i$ ending with the arc $a_1 = a_i$ have this arc in common. In the following, we therefore assume $\mathcal{C} = \{\{1, i\} \colon a_1 = a_i, i \neq 1\} \subseteq \mathcal{I}$ and define $\tilde{\mathcal{I}} = \mathcal{I} \setminus \mathcal{C}$. We claim that the truth value of $((a_1, \dots, a_k), \mathcal{I})$ is the disjunction

$$x((a_1, \dots, a_k), \mathcal{I}) = \bigvee \left\{ x((\tilde{a}, a_2, \dots, a_k), \tilde{\mathcal{I}}) : \tilde{a} \in \delta^{\text{in}}(\alpha(a_1)) \right\} \tag{5.1}$$

of truth values of smaller states. Note that all states in the disjunction are indeed smaller as $\nu(\alpha(\tilde{a})) < \nu(\omega(\tilde{a})) = \nu(\alpha(a_1))$ due to fact that $\nu$ is a topological ordering. We now prove the correctness.

First, suppose that $x((a_1, \dots, a_k), \mathcal{I})$ is `true`. Thus, there exist almost disjoint paths $P_1, \dots, P_k$ with intersection pattern $\mathcal{I}$ and last arcs $a_1, \dots, a_k$. The paths remain almost disjoint when removing the last arc $a_1$ from $P_1$. This removal changes the intersection pattern from $\mathcal{I}$ to $\tilde{\mathcal{I}}$. This shows that $x((\tilde{a}, a_2, \dots, a_k), \tilde{\mathcal{I}})$ is `true` for $\tilde{a}$ being the penultimate arc on the path $P_1$. As $\tilde{a} \in \delta^{\text{in}}(\alpha(a_1))$, this value is contained in the disjunction from Equation (5.1).

Now, suppose that $x((\tilde{a}, a_2, \dots, a_k), \tilde{\mathcal{I}})$ is `true` for an arc $\tilde{a} \in \delta^{\text{in}}(\alpha(a_1))$ and let $\tilde{\mathcal{I}}$ be as defined above. Again, there are almost disjoint paths $P_1, P_2, \dots, P_k$ with last arcs $\tilde{a}, a_2, \dots, a_k$ and intersection pattern $\tilde{\mathcal{I}}$. By the choice of the arc $a_1$, we obtain that $a_1$ is the last arc of a path $P_i$ whenever it is contained in $P_i$. To see this, recall that $a_1$ is among the arcs $\{a_1, \dots, a_k\}$ one whose start-vertex has the largest value $\nu(\alpha(a_1))$ in the topological ordering. If $a_1$ is contained in a path $P_i$ but not the last arc $a_1 \neq a_i$, we have $\nu(\alpha(a_i)) > \nu(\alpha(a_1))$ which is a contradiction.

Since $\tilde{\mathcal{I}} \cap \mathcal{C} = \varnothing$, the path $P_1$ shares no arc with another path that ends with $a_1$. And because $a_1$ can only be the last arc of a path $P_i$, we can extend $P_1$ by the arc $a_1$ while maintaining that the paths are almost disjoint. Moreover, the new intersection pattern is $\mathcal{I}$ as

$$\mathcal{I} = \tilde{\mathcal{I}} \cup \mathcal{C} = \tilde{\mathcal{I}} \cup \{\{1, i\} \colon a_1 = a_i, i \neq 1\} = \tilde{\mathcal{I}} \cup \{\{1, i\} : a_1 \in P_i, i \neq 1\}.$$

This shows the correctness of the recursion from Equation (5.1). Hence, we can compute the truth values of all states in polynomial time and the claim follows. □

**Theorem 5.3.** For constant $k$, ADP is polynomial time solvable.

*Proof.* We prove the claim by converting $G = (V, A)$ into a directed acyclic graph $G'$ and by adapting the dynamic program from the proof of Theorem 5.2 to the new situation. To this end, let $n = |V|$.

The vertex set of $G'$ consists of $n$ copies $v_1, \dots, v_n$ for every vertex $v \in V$, and we call the vertices $\{v_i : v \in V\}$ the *i-th layer* of $G'$. For every arc $uv \in A$ we add the $n - 1$ arcs $u_{i-1}v_i$ for $i \in \{2, \dots, n\}$ to $G'$, which we call *copies* of $uv$. Since all arcs in $G'$ lead exactly one layer up, the graph $G'$ constructed so far is acyclic. Furthermore, we add an additional vertex $t'$ to $G'$, which we connect with arcs $t_it'$ for $i \in \{1, \dots, n\}$. Note that $G'$ remains acyclic. In order to ensure Assumption 4.9 we restrict $G'$ to those vertices and arcs that are reachable from $s_1$ and from which we can reach $t'$.

The basic idea is now to find $k$ almost disjoint $s_1$-$t'$-paths in $G'$ and translate these back to $s$-$t$-paths in the original graph $G$. For this we ignore the last vertex $t'$ and replace every other vertex $v_i$ on such a path by the corresponding vertex $v \in V$. In this way, however, almost disjoint paths in $G'$ need not remain almost disjoint in $G$. For this to be the case, we have to identify all copies of an arc: for every two paths that we choose in $G'$ there must be at most one arc $uv \in A$ of which both paths contain a copy. If this is the case, we call the paths *almost copy-disjoint*.

To achieve this, we have to slightly modify the dynamic program from the proof of Theorem 5.2. More precisely, we update the definition of the Boolean value $x((a_1, \dots, a_k), \mathcal{I})$. Instead of assuming the $s$-$\omega(a_i)$-paths $P_i$ to be almost disjoint, we now require that they are almost copy-disjoint. Accordingly, we have to update the interpretation of the intersection pattern: we now have $\{i, j\} \in \mathcal{I}$ if and only if the paths $P_i$ and $P_j$ both contain a copy of the same arc.

Consequently, we have to amend the recursion. A state must be `false` not only if $a_1 = a_i$ and $\{1, i\} \notin \mathcal{I}$, but even if $a_1$ and $a_i$ are copies of the same arc and $\{1, i\} \notin \mathcal{I}$. This also entails a slightly different definition of $\mathcal{C}$:

$$\mathcal{C} = \{\{1, i\} : a_1 \text{ and } a_i \text{ are copies of the same arc}, i \neq 1\}.$$

Overall, the size of $G'$ is polynomial in the size of $G$, it can be constructed in polynomial time, and all modifications in the dynamic program induce only polynomial overhead. □

## 5.2 Polynomial-Size IP Formulation

With Program (4.1) we already have an integer programming formulation for MaxADP. However, this formulation has an exponential number of variables. With Program (5.2) we now provide an integer programming formulation of polynomial size.

$$
\begin{aligned}
\max \quad & \sum_{i=1}^{k} c(i) & & \text{(5.2a)}\\
\text{s.t.} \quad & c(i) = \sum_{a\in\delta^{\text{out}}(s)} x(i,a) & \forall\, i \in \{1,\dots,k\} & \quad \text{(5.2b)}\\
& \sum_{a\in\delta^{\text{in}}(v)} x(i,a) = \sum_{a\in\delta^{\text{out}}(v)} x(i,a) & \forall\, i \in \{1,\dots,k\}\ \forall\, v \in V\setminus\{s,t\} & \quad \text{(5.2c)}\\
& x(i,a) + x(j,a) - b(i,j,a) \le 1 & \forall\, 1 \le i < j \le k\ \forall\, a \in A & \quad \text{(5.2d)}\\
& \sum_{a\in A} b(i,j,a) \le 1 & \forall\, 1 \le i < j \le k & \quad \text{(5.2e)}\\
& c(i) \in \{0,1\} & \forall\, i \in \{1,\dots,k\} & \quad \text{(5.2f)}\\
& x(i,a) \in \{0,1\} & \forall\, i \in \{1,\dots,k\}\ \forall\, a \in A & \quad \text{(5.2g)}\\
& b(i,j,a) \in \{0,1\} & \forall\, 1 \le i < j \le k\ \forall\, a \in A & \quad \text{(5.2h)}
\end{aligned}
$$

For an upper bound $k$ on the number of almost disjoint paths, we have binary variables $c(i)$ stating whether an $i$-th path is chosen. Consequently, Objective (5.2a) aims to maximize the number of chosen paths. If we choose an $i$-th path, we specify its arcs with the binary variables $x(i,a)$. Such a variable indicates for an arc $a$ whether it is contained in the $i$-th path. Constraints (5.2b) and (5.2c) ensure that these arcs form indeed an $s$-$t$-path whenever $c(i) = 1$. To guarantee that no two paths have more than one arc in common, Program (5.2) uses the binary variables $b(i,j,a)$. If an arc $a$ is chosen in the $i$-th as well as in the $j$-th path, Constraint (5.2d) enforces $b(i,j,a) = 1$. Constraint (5.2e) thus limits the number of common arcs for every two paths to one.

This integer programming formulation requires an upper bound $k$ on the maximum number of almost disjoint $s$-$t$-paths in $G$. If this bound is polynomial in the instance size, also Program (5.2) is polynomially large. Fortunately, we can bound the number of almost disjoint paths by $\mathcal{O}(|A|^2)$. One option is to pair every arc $sv \in \delta^{\text{out}}(s)$ with every arc $vu \in \delta^{\text{out}}(v)$, which results in $k = \sum_{sv\in\delta^{\text{out}}(s)} |\delta^{\text{out}}(v)| \in \mathcal{O}(|A|^2)$. Another possibility is to combine every arc $sv \in \delta^{\text{out}}(s)$ leaving the source $s$ with an arc $ut \in \delta^{\text{in}}(t)$ entering the target $t$ resulting in $k = |\delta^{\text{out}}(s)| \cdot |\delta^{\text{in}}(t)| \in \mathcal{O}(|A|^2)$. Using one of these upper bounds results in $\mathcal{O}(|A|^5)$ variables and constraints.

## 5.3 NP-Completeness

Although we can solve ADP for constant $k$ in polynomial time, it is **NP**-complete in general. This is stated in Theorem 5.4, which we prove in this section.

**Theorem 5.4.** ADP is **NP**-complete, even on acyclic graphs.

ADP is contained in **NP** since we can check in polynomial time whether $k$ paths are almost disjoint and since we can polynomially bound the maximum number of almost disjoint $s$-$t$-paths as shown in the previous section. In the rest of the section, we prove the **NP**-hardness of ADP by a reduction from the **NP**-complete [GJ79] independent set problem IndSet. For this purpose, let an instance of the independent set problem be given by an undirected graph $H = (V_H, E_H)$. After constructing a directed acyclic graph $G = (V, A)$ we show that $H$ has an independent set of size $k$ if and only if there are $2 \cdot |E_H| + k$ almost disjoint $s$-$t$-paths in $G$.

### Graph Construction

**The Gadget** The basic component to construct the graph $G$ is the edge gadget depicted in Figure 5.1. Such a gadget $\text{gad}(uv)$ corresponds to an edge $uv \in E_H$ and has four inputs: two labeled $u$ and $v$ corresponding to the end vertices of the edge and two auxiliary inputs $h_1$ and $h_2$. Analogously, the gadget also has four outputs: $u'$ and $v'$ as well as $h'_1$ and $h'_2$. In addition, it contains ten interior vertices, which we name and connect as drawn in Figure 5.1. When using this gadget in the graph construction, we rearrange its in- and output vertices as shown in Figure 5.2.

**The Graph** The graph $G = (V, A)$ of the ADP instance corresponding to the graph $H = (V_H, E_H)$ is drawn in Figure 5.3. It consists of a gadget $\text{gad}(e)$ for every edge $e \in E_H$ and additional vertices $V_H \cup \{s, t, v_V, v_E\}$. The source $s$ is connected with $v_V$ and $v_E$ and the vertex $v_V$ has outgoing arcs to all $v \in V_H$. To every auxiliary input of a gadget we have an arc from $v_E$. From every auxiliary output of a gadget there is an arc to the target $t$.

Finally, we explain how the vertex inputs of the gadgets are connected. To this end, sort the edges $E_H = \{e_1, \ldots, e_m\}$ arbitrarily. In the graph $G$, every vertex $u \in V_H$ is connected to the target $t$ by a path $P_u$ that starts with $(s, v_V, u)$ and passes through every gadget $\text{gad}(e)$ corresponding to an incident edge $e \in \delta_H(u)$. For $\ell = \deg_H(u)$ we choose $j_1 < \cdots < j_\ell$ such that $\delta_H(u) = \{e_{j_1}, \ldots, e_{j_\ell}\}$. We connect $u$ with the input of $\text{gad}(e_{j_1})$ that is labeled $u$. Its output $u'$ is connected with the input $u$ of $\text{gad}(e_{j_2})$ and so on. Finally, the output $u'$ of the last gadget $\text{gad}(e_{j_\ell})$ has an arc to the target. If the vertex $u \in V_H$ has no incident edge, we introduce the direct arc $ut$.

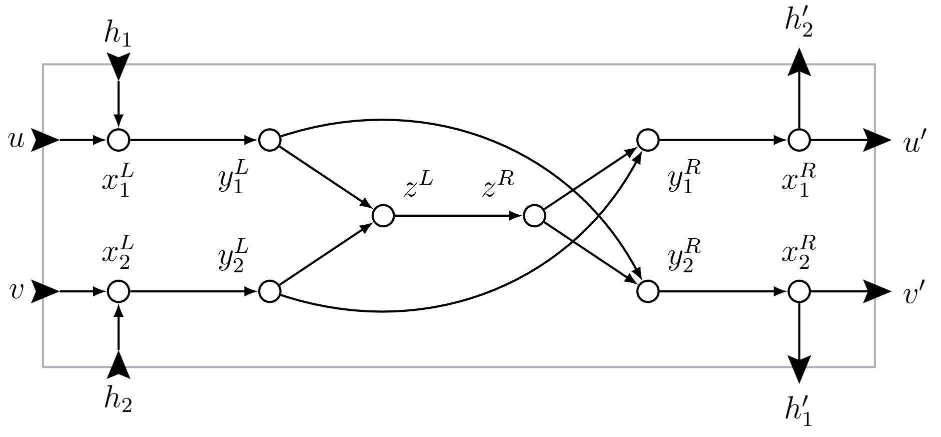


Figure 5.1: The gadget gad($uv$) of an edge $uv \in E_H$.

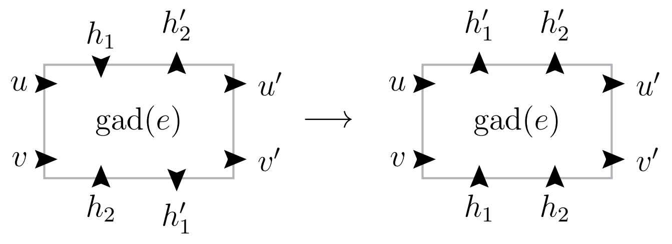


Figure 5.2: The rearrangement of the in- and outputs of a gadget from Figure 5.1 in order to simplify the resulting graph in Figure 5.3.

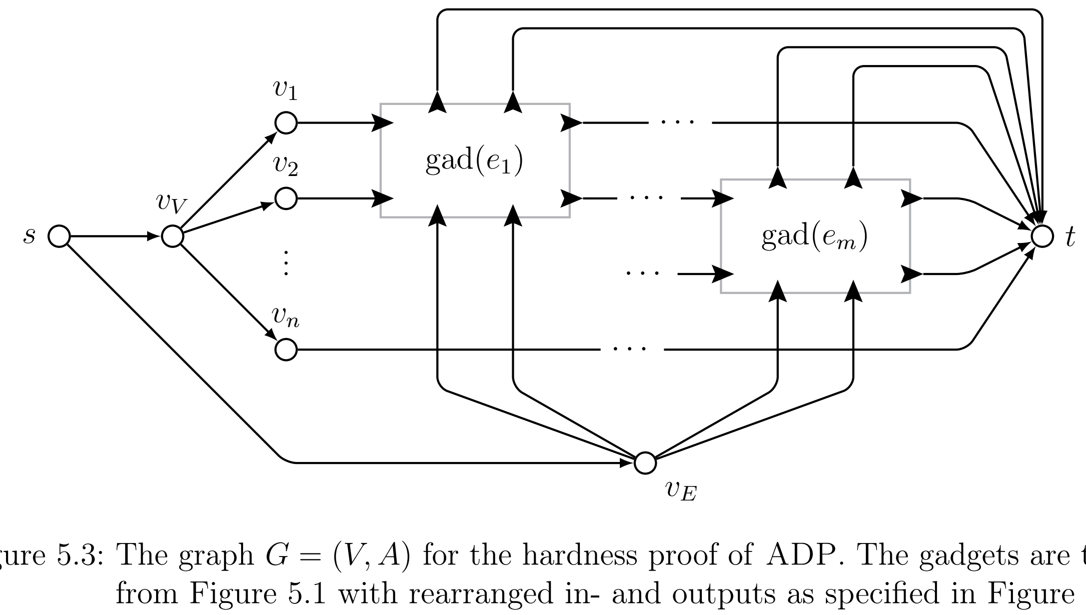


Figure 5.3: The graph $G = (V, A)$ for the hardness proof of ADP. The gadgets are those from Figure 5.1 with rearranged in- and outputs as specified in Figure 5.2.

**Definition 5.5** (Auxiliary and Vertex Paths)**.** Every $s$-$t$-path in $G$ either starts with the arc $sv_V$ or with the arc $sv_E$. Those starting with $sv_V$ are *vertex paths* and those starting with $sv_E$ are *auxiliary paths.*

In the following, we prove that $H$ contains an independent set of size $k$ if and only if there are $2m + k$ almost disjoint paths in $G$. The direction from an independent set to almost disjoint paths is relatively simple whereas the converse is a bit more involved.

## From an Independent Set to Almost Disjoint Paths

**Lemma 5.6.** Given an independent set $U \subseteq V_H$ in $H$ of size $|U| = k$, there are $2m + k$ almost disjoint $s$-$t$-paths in $G$.

*Proof.* We construct $2m + k$ almost disjoint $s$-$t$-paths, from which $k$ are vertex paths corresponding to the vertices in $U$ and the remaining $2m$ are auxiliary paths.

The auxiliary paths are obtained by extending the $h_1$-$h_1'$- and $h_2$-$h_2'$-paths visualized in Figure 5.4 of all gadgets $\mathrm{gad}(e)$, $e \in E_H$. They have the first arc $sv_E$ in common and are disjoint afterwards.

For $u \in U$ we choose the unique $s$-$t$-path in $G$ that starts with $(s, v_V, u)$ and uses all $u$-$u'$-paths through gadgets $\mathrm{gad}(e)$ of incident edges $e \in \delta_H(u)$ as well as the arcs connecting these. We denote this path by $P_u$. Since $U \subseteq V_H$ is an independent set in $H$ and since the gadgets correspond to edges in $H$, there is no gadget $\mathrm{gad}(uv)$ with $\{u, v\} \subseteq U$. Thus, for every gadget, we choose at most one vertex path passing through it. This implies that also all chosen vertex paths have the first arc $sv_V$ in common and are disjoint afterwards.

A vertex path $P_u$ and an auxiliary path have exactly one arc in common, if the auxiliary path passes through a gadget corresponding to an edge that is incident to $u$. Otherwise, the two paths are disjoint. Hence, the $2m + k$ chosen $s$-$t$-paths are almost disjoint. □

## From Almost Disjoint Paths to an Independent Set

In the following, let $\mathcal{Q}$ be a set of $2m + k$ almost disjoint $s$-$t$-paths in $G$ among which the number of auxiliary paths is maximized. Note that we need not to be able to find such a set $\mathcal{Q}$ constructively but only have to know that it exists. Nevertheless, the way we build on this assumption below will still open up possibilities for constructing it. We assume $k \geq 0$ as we can choose $2m$ auxiliary paths as described in the proof of Lemma 5.6.

**Assumption 5.7.** No set of $2m+k$ almost disjoint $s$-$t$-paths in $G$ contains more auxiliary paths than $\mathcal{Q}$.

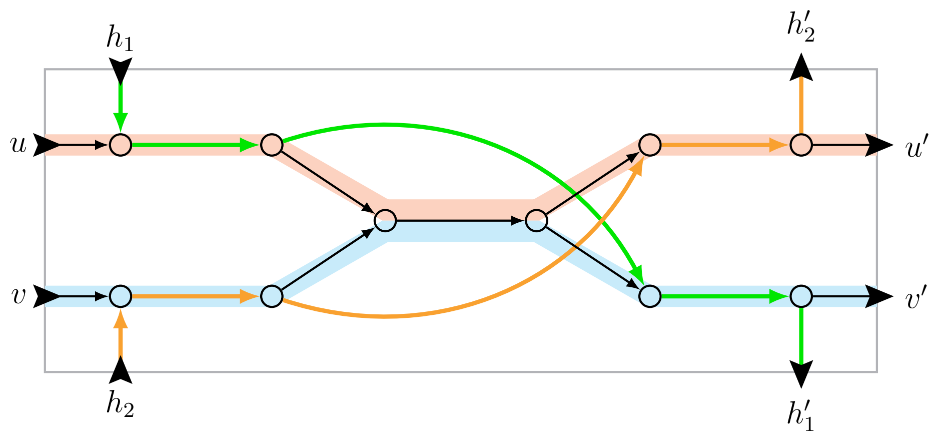


Figure 5.4: A gadget as in Figure 5.1 with four different paths passing through it: the unique $u$-$u'$-path in red, the unique $v$-$v'$-path in blue, an $h_1$-$h_1'$-path in green, and an $h_2$-$h_2'$-path in orange.

The following three lemmas provide structural results of the paths in $\mathcal{Q}$. They allow us to prove in Lemma 5.11 the counterpart of Lemma 5.6, which completes the proof of Theorem 5.4.

**Lemma 5.8.** Without loss of generality we can assume that every auxiliary path in $\mathcal{Q}$ leaves the gadget it enters first via an auxiliary output.

*Proof.* Let $P \in \mathcal{Q}$ be an auxiliary path, let $\operatorname{gad}(e)$ be the gadget it enters first, and suppose that $P$ leaves $\operatorname{gad}(e)$ via a vertex output $u'$. By the construction of the graph, there is a single arc leaving $u'$. This arc either points to the vertex input of another gadget or to the target.

We first consider the case that the arc leaving $u'$ points to a vertex input $\tilde{u}$ of another gadget $\operatorname{gad}(\tilde{e})$. In this situation, which is depicted in Figure 5.5, the path $P$ enters $\operatorname{gad}(\tilde{e})$ via $\tilde{u}$ directly after leaving $\operatorname{gad}(e)$ via $u'$. No auxiliary path leaves $\operatorname{gad}(e)$ via $h_2'$ and no auxiliary path enters $\operatorname{gad}(\tilde{e})$ via $\tilde{h}_1$. Otherwise, such a path has not only the arc $sv_E$ in common with $P$, but also either $y_1^R x_1^R$ or $\tilde{x}_1^L \tilde{y}_1^L$. Thus, we can replace $P$ in $\mathcal{Q}$ by two auxiliary paths: one that equals $P$ until vertex $x_1^R$ but then continues along $(x_1^R, h_2', t)$ and the other starting with $(s, v_E, \tilde{h}_1', \tilde{x}_1^L)$ and following $P$ from $\tilde{x}_1^L$ on. To preserve the amount of paths in $\mathcal{Q}$, we remove a vertex path from $\mathcal{Q}$ in return.

Note that the paths from $\mathcal{Q}$ remain almost disjoint after this modification, except if there is a vertex path leaving $\operatorname{gad}(e)$ via $h_2'$. However, if this is the case we can simple remove this vertex path. Also note that $\mathcal{Q}$ contains at least one vertex path since $k \geq 0$ and because $\mathcal{Q}$ contains at most $2m-1$ auxiliary paths: $\deg^{\text{out}}(v_E) = 2m$ and no auxiliary path uses the arc $v_E\tilde{h}_1$. Thus, the replacement of $P$ in $\mathcal{Q}$ contradicts Assumption 5.7 such that this case cannot occur.

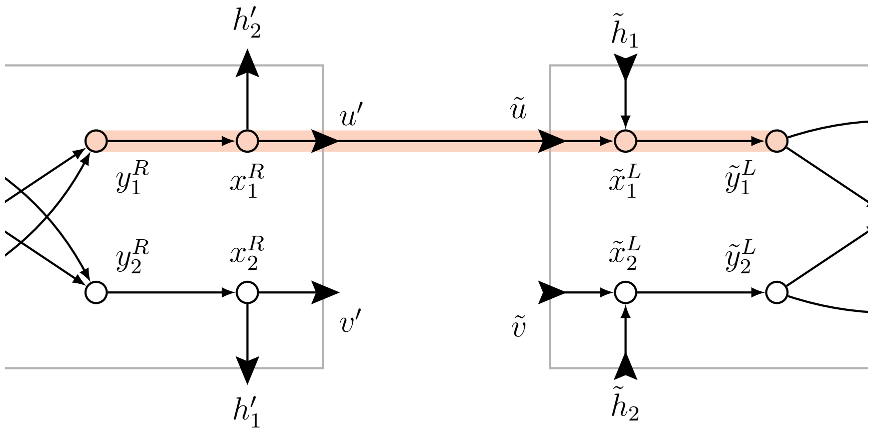


Figure 5.5: A path that leaves gad($e$) via its vertex output $u'$ and enters gad($\tilde{e}$) via its vertex input $\tilde{u}$ uses at least the red marked arcs.

We now consider the remaining case, in which the arc leaving $u'$ directly points to the target $t$. In this situation, $P$ ends with $(y_1^R, x_1^R, u', t)$ and we can modify $P$ by using $(y_1^R, x_1^R, h'_2, t)$ instead. As argued in the first case, another path in $\mathcal{Q}$ leaving gad($e$) via $h'_2$ has to be a vertex path. If it exists, we can modify it by changing its end from $(y_1^R, x_1^R, h'_2, t)$ to $(y_1^R, x_1^R, u', t)$.

Hence, these modifications reduce the number of auxiliary paths leaving a gadget via a vertex output by one. Formally, the claim follows by induction. □

**Lemma 5.9.** Let gad($e$) be a gadget that is passed through by exactly one vertex path $P$ from $\mathcal{Q}$. If $P$ enters gad($e$) via a vertex input $u$, it leaves gad($e$) via the corresponding vertex output $u'$.

*Proof.* If $P$ leaves gad($e$) via an auxiliary output, Lemma 5.8 implies that $\mathcal{Q}$ contains at most one auxiliary path passing through gad($e$). In this case, we can remove all paths passing through gad($e$) from $\mathcal{Q}$ and replace them by the same amount of auxiliary paths. This increases the amount of auxiliary paths in $\mathcal{Q}$ contradicting Assumption 5.7.

Next, suppose that $P$ leaves gad($e$) via the vertex output $v'$. In this case, it must definitely use the arcs that are marked in red in Figure 5.6. The only chance for an almost disjoint auxiliary path $P'$ entering gad($e$) via $h_1$ is to use the $h_1$-$h'_2$-path whose arcs are green in Figure 5.6. However, every other auxiliary path entering gad($e$) must share an arc of gad($e$) with $P'$. Thus, they are not almost disjoint and $\mathcal{Q}$ contains again at most one auxiliary path passing through gad($e$). As in the first case we can replace all paths through gad($e$), thereby increasing the amount of auxiliary paths in $\mathcal{Q}$, and again contradicting Assumption 5.7. □

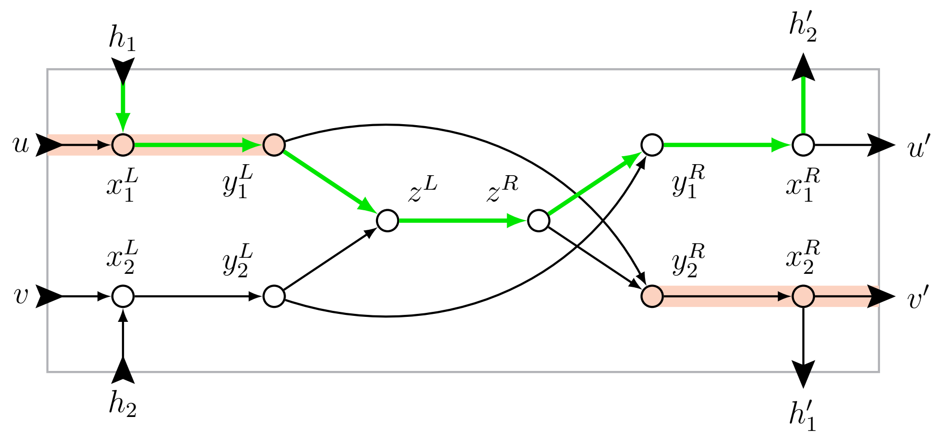


Figure 5.6: A vertex path entering a gadget gad($e$) via a vertex input $u$ and leaving it via the “wrong” vertex output $v'$ has to use at least the red marked arcs. In this situation, an auxiliary path entering gad($e$) via $h_1$ must follow the green arcs. A visualization for the proof of Lemma 5.9.

**Lemma 5.10.** There is no gadget through which two vertex paths of $\mathcal{Q}$ pass.

*Proof.* Suppose there is a gadget gad($e$) that is passed through by two vertex paths. Denote these paths by $P_u$ and $P_v$. Since they already have the arc $sv_V$ in common, they are disjoint in gad($e$). Hence, the path $P_u$ entering gad($e$) via $u$ has to use $y_2^R x_2^R$ and the path $P_v$ entering gad($e$) via $v$ has to use $y_1^R x_1^R$.

Furthermore, there is also an auxiliary path passing through gad($e$) because otherwise we could replace one of the two vertex paths by an auxiliary path contradicting Assumption 5.7. Similarly to the proof of Lemma 5.9, this auxiliary path has to be either an $h_1$-$h_2'$-path or an $h_2$-$h_1'$-path inside gad($e$). By symmetry we assume without loss of generality that it is an $h_1$-$h_2'$-path. Hence, the situation is as depicted in Figure 5.7 (the path $P_u$ can leave gad($e$) either via $v'$ or via $h_1'$).

We now construct a new vertex path $P$ that replaces $P_u$ and $P_v$ in $\mathcal{Q}$. This path first uses $P_u$ until $y_1^L$ in gad($e$). From thereon it uses $(y_1^L, z^L, z^R, y_1^R)$ and then continues like $P_v$. Since all vertex paths are disjoint after $v_V$, the new path $P$ has only the arc $sv_V$ in common with any of the remaining vertex paths. Moreover, it has at most one arc in common with any auxiliary path outside of gad($e$) since this was already the case for $P_u$ and $P_v$. We also replace the auxiliary $h_1$-$h_2'$-path by an auxiliary $h_1$-$h_1'$-path and an auxiliary $h_2$-$h_2'$-path. The resulting paths passing through gad($e$) are visualized in Figure 5.8.

After these modifications, $\mathcal{Q}$ contains the same number of paths but the amount of auxiliary paths increases by one. This contradicts Assumption 5.7. □

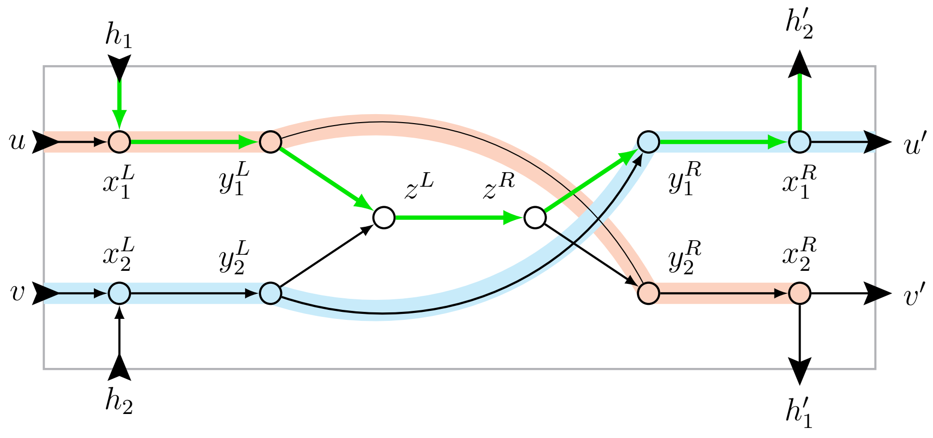


Figure 5.7: Two vertex paths passing a gadget gad($e$). The path $P_u$ entering via $u$ contains at least the red marked arcs, the path $P_v$ entering via $v$ the arcs marked blue. An additional auxiliary path is drawn with green arcs. A visualization for the proof of Lemma 5.10.

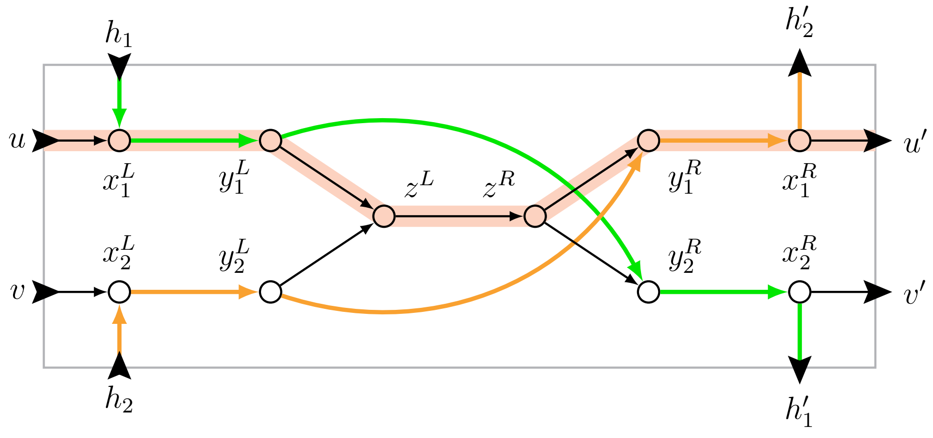


Figure 5.8: The result of the modifications in Lemma 5.10. The two vertex paths and the auxiliary path from Figure 5.7 are replaced by the vertex path marked red and the two auxiliary paths drawn in green and orange.

Using Lemmas 5.8 to 5.10 we are now able to prove in Lemma 5.11 the counterpart of Lemma 5.6 and thus complete the proof that ADP is **NP**-complete.

**Lemma 5.11.** Given $2m + k$ almost disjoint $s$-$t$-paths in $G$, there is an independent set $U \subseteq V_H$ in $H$ of size $|U| = k$.

*Proof.* We choose a set $\mathcal{Q}$ of $2m + k$ almost disjoint $s$-$t$-paths in $G$ that fulfills Assumption 5.7. By Lemma 5.8 we also assume that every auxiliary path in $\mathcal{Q}$ passes through exactly one gadget.

We define $U \subseteq V_H$ to be the set of vertices that are contained in a vertex path of $\mathcal{Q}$. We first prove that $U$ is an independent set in $H$.

Lemma 5.10 implies that for every gadget there is at most one vertex path in $\mathcal{Q}$ that passes through this gadget. If this is the case, Lemma 5.9 states that this vertex path enters the gadget via a vertex input $u$ and leaves it via the corresponding vertex output $u'$. Thus, a vertex path starting with $(s, v_V, u)$ passes through a gadget $\text{gad}(e)$ if and only if $e \in \delta_H(u)$. Because no two vertex paths from $\mathcal{Q}$ pass through the same gadget, we obtain that $U$ is indeed an independent set.

We complete the proof by showing that $U$ contains $k$ elements. Every gadget is used by at most one vertex path, see Lemma 5.10. Moreover, such a vertex path leaves every gadget via the correct vertex output, see Lemma 5.9 again. Thus, we can additionally choose two auxiliary paths passing through every gadget. Furthermore, since $\mathcal{Q}$ fulfills Assumption 5.7, it contains $2m$ auxiliary paths. And since there can only be at most $2m$ almost disjoint auxiliary paths, $\mathcal{Q}$ contains exactly $k$ vertex paths. Because these are also almost disjoint, they contain distinct vertices $u \in V_H$ showing $|U| = k$. $\square$

## 5.4 APX-Hardness

In this section, we lift the **NP**-hardness proof for ADP from the previous section to an **APX**-hardness proof for MAXADP. This shows that the almost disjoint paths problem is even hard to approximate within a constant factor.

**Theorem 5.12.** MAXADP is **APX**-hard, even on acyclic graphs.

*Proof.* We essentially show that the Karp reduction that proves the **NP**-hardness of ADP in Section 5.3 is actually an $L$-reduction if we restrict the graph $H$ (in which we search for an independent set) to have maximum degree $\Delta(H) \leq 3$. We may do this since the independent set problem remains **APX**-complete on graphs whose maximum degree is bounded by a constant $B \geq 3$, see [ACG+99, Problem GT23], [PY91], and [BF95]. We call this problem INDSET-$B$ and restrict ourselves to INDSET-3 in the following.

In order to prove that IndSet-3 $L$-reduces to MaxADP we are guided by the definition of an $L$-reduction from [PY91]: we provide polynomial-time computable functions $f$ and $g$ as well as constants $\alpha, \beta > 0$ such that

(a) the function $f$ maps instances of IndSet-3 to instances of MaxADP such that $OPT_{\text{MaxADP}}(f(H)) \leq \alpha \cdot OPT_{\text{IndSet-3}}(H)$ for every IndSet-3 instance $H$, and

(b) the function $g$ maps every set $\mathcal{Q}$ of almost disjoint $s$-$t$-paths in $f(H)$ to an independent set $U$ of $H$ with $OPT_{\text{IndSet-3}}(H) - |U| \leq \beta \cdot (OPT_{\text{MaxADP}}(f(H)) - |\mathcal{Q}|)$.

Here, $OPT_{\text{IndSet-3}}(H)$ and $OPT_{\text{MaxADP}}(f(H))$ denote the maximum size of an independent set in $H$ and the maximum number of almost disjoint $s$-$t$-paths in $f(H)$. Let $H = (V_H, E_H)$ be an undirected graph as in Section 5.3. We write $n = |V_H|$ and $m = |E_H|$.

We first show that (a) holds for $\alpha = 13$ and for the function $f$ representing the graph construction from Section 5.3. This construction only requires polynomial time and, thus, the function $f$ is polynomial-time computable. A greedy algorithm that iteratively chooses an arbitrary vertex and removes it with its at most 3 neighbors results in an independent set with at least $n/4$ vertices. Hence, we have $OPT_{\text{IndSet-3}}(H) \geq n/4$. With $m \leq 3n/2$ (due to $\Delta(H) \leq 3$) this results in

$$m \leq 6 \cdot OPT_{\text{IndSet-3}}(H). \tag{5.3}$$

From Lemmas 5.6 and 5.11 we obtain

$$OPT_{\text{MaxADP}}(f(H)) = 2m + OPT_{\text{IndSet-3}}(H). \tag{5.4}$$

Using (5.3) in Equation (5.4) we have $OPT_{\text{MaxADP}}(f(H)) \leq 13 \cdot OPT_{\text{IndSet-3}}(H)$.

We now show that (b) holds for $\beta = 1$ and for the function $g$ that represents the procedure from the proof of Lemma 5.11. This procedure transforms a set $\mathcal{Q}$ of $2m + k$ almost disjoint paths in $f(H)$ into an independent set $U$ of size $k$ in $H$. Equation (5.4) yields

$$\begin{aligned} OPT_{\text{MaxADP}}(f(H)) - |\mathcal{Q}| &= 2m + OPT_{\text{IndSet-3}}(H) - (2m + k) \\ &= OPT_{\text{IndSet-3}}(H) - k \\ &= OPT_{\text{IndSet-3}}(H) - |U|. \end{aligned}$$

If $|\mathcal{Q}| \leq 2m$, we can choose $U = \varnothing$. Using again Equation (5.4) in an analogous computation results in

$$OPT_{\text{MaxADP}}(f(H)) - |\mathcal{Q}| \geq OPT_{\text{IndSet-3}}(H) - |U|.$$

What remains to prove is that $g$ is polynomial-time computable. This is not obvious since the proof of Lemma 5.11 requires a set of almost disjoint paths that satisfies Assumption 5.7 instead of an arbitrary one. Contrary to our comment before Assumption 5.7 on Page 44, we now need to be able to transform an arbitrary set $\mathcal{Q}$ of almost disjoint $s$-$t$-paths in polynomial time into one of the same cardinality that satisfies Assumption 5.7. However, the proofs of Lemmas 5.8 to 5.10 are constructive such that they provide a way to perform exactly this task in polynomial time. □

## Conclusion

We analyzed the complexity of the almost disjoint paths problem, and one of the key findings was that it is **NP**-complete, even on acyclic graphs. In addition, we showed that the presented reduction actually is an $L$-reduction and derived the **APX**-hardness of MaxADP. Furthermore, we explained how to find two almost disjoint paths efficiently, and we established a dynamic program that allows solving ADP for constant $k$ in polynomial time. In the following chapter, we focus on the complexity of the separating by forbidden pairs problem.

# Chapter 6

# Separating by Forbidden Pairs

*As the main result of this chapter, we prove that the Separating by Forbidden Pairs (*SFP*) problem is $\Sigma_2^p$-complete. Additionally, we provide a heuristic for* MinSFP *and analyze its capabilities and limits.*

## Assumptions and Notes

Throughout this chapter, $G = (V, A)$ denotes a directed graph and $s, t \in V$ are distinct source and target vertices in this graph. The following assumptions apply.

**Only Relevant Arcs and Vertices** → see Assumption 4.9 on Page 28
Every arc and every vertex is contained in an $s$-$t$-path.

**No Direct Arc** → see Assumption 4.10 on Page 28
The graph $G$ does not contain the arc $st$.

We formally defined SFP in Problem 4.4 on Page 26. Recall that it asks for a set $\mathcal{A}$ of arc pairs such that every $s$-$t$-path contains both arcs of at least one pair of $\mathcal{A}$.

Large parts of this chapter are joint work with Oliver Bachtler and Sven O. Krumke, which is already published in [BBK22].

## 6.1 Upper Bounds and Examples

In Lemma 4.16 on Page 33 we saw that we can solve SFP in polynomial time if the graph $G$ contains an $s$-$t$-cut $(S, T)$ with a single outgoing arc $\delta^{\text{out}}(S) = \{uv\}$. The proof was based on the fact that every $s$-$t$-path must use this arc and on the weak duality of ADP and SFP: we get an optimal set of separating pairs by pairing $uv$ with every arc of a minimum $s$-$u$- or $v$-$t$-cut depending on which of the two is smaller. In Algorithm 6.1 we extend this idea to a heuristic for MinSFP. Instead of an $s$-$t$-cut with a single outgoing arc we now allow to choose an arbitrary $s$-$t$-cut $(S, T)$ in $G$. For every arc $uv \in \delta^{\text{out}}(S)$ we proceed as before and add pairs that suffice to separate all $s$-$t$-paths using $uv$.

**Algorithm 6.1:** Minimum Cut Heuristic

**Input:** A directed graph $G = (V, A)$ with source and target vertices $s, t \in V$.
**Output:** A set $\mathcal{A} \subseteq \binom{A}{2}$ of arc pairs separating $s$ and $t$ in $G$ in terms of SFP.

```
1  Initialize $\mathcal{A} = \varnothing$.
2  Choose an $s$-$t$-cut $(S, T)$.
3  for every arc $a = uv \in \delta^{\text{out}}(S)$ in the cut do
4      if $u \neq s$ then
5          Determine a minimum $s$-$u$-cut $(S_u, T_u)$.
6      end
7      if $v \neq t$ then
8          Determine a minimum $v$-$t$-cut $(S_v, T_v)$.
9      end
10     Let $(S_a, T_a)$ be the one of the (at most) two cuts with smaller cardinality.
11     Add the pairs $\{(a, a') : a' \in \delta^{\text{out}}(S_a)\}$ to $\mathcal{A}$.
12 end
13 return $\mathcal{A}$
```

**Lemma 6.1.** The set $\mathcal{A}$ returned by Algorithm 6.1 is a solution for MinSFP, that is, it contains at least one pair on every $s$-$t$-path in $G$.

*Proof.* Let $P$ be an $s$-$t$-path. Since $(S, T)$ is an $s$-$t$-cut, there exists at least one arc $a = (u, v) \in A(P) \cap \delta^{\text{out}}(S)$ that is contained in the path and in the cut. Since $(S_a, T_a)$ is either an $s$-$u$-cut or a $v$-$t$-cut, there is at least one arc $a' \in A(P) \cap \delta^{\text{out}}(S_a)$. Thus, the path $P$ contains both arcs of the pair $\{a, a'\} \in \mathcal{A}$. □

Analogous to Lemma 4.16 we prove in Lemma 6.2 that Algorithm 6.1 computes an optimum separating set of arc pairs if the graph has an $s$-$t$-cut with a single outgoing arc and if we choose this cut at the start of the algorithm.

**Lemma 6.2.** If a $G$ has an $s$-$t$-cut $(S, T)$ with a single outgoing arc $\delta^{\text{out}}(S^*) = \{uv\}$, Algorithm 6.1 with choosing $(S, T)$ in Line 2 computes an optimal solution of MinSFP.

*Proof.* By Menger's Theorem there exist $|(S_u, T_u)|$ arc-disjoint $s$-$u$-paths and $|(S_v, T_v)|$ arc-disjoint $v$-$t$-paths, where $S_u$ and $S_v$ are as defined in Lines 5 and 8 of Algorithm 6.1 [see Die00, Theorem 3.3.5]. Hence, there are $|(S_{uv}, T_{uv})|$ many paths that only have the arc $uv$ in common. For every such path we need a separate forbidden pair. □

In the following, we examine capabilities and limits of this heuristic. First, Example 6.3 provides an instance for which Algorithm 6.1 is able to find the unique optimal solution whose both pairs are contained in one path if the "right" cut is chosen in Line 2. In general, an obvious choice for this initial cut would be a minimum $s$-$t$-cut. However, Example 6.4 points out that this choice does not necessarily lead to an optimal solution. In fact, in Example 6.7 we present instances on which this heuristic cannot find an optimal solution no matter which initial $s$-$t$-cut we choose. This even holds for an improved version of the heuristic that we develop in Example 6.5 and Lemma 6.6.

**Example 6.3.** Some paths may contain multiple forbidden pairs of an optimal solution of SFP. Such a situation is depicted in Figure 6.1. The unique optimal solution $\mathcal{A} = \{\{sv_1, v_1v_2\}, \{v_3v_4, v_4t\}\}$ consists of two forbidden pairs. All four involved arcs are contained in the path $P = (s, v_1, v_2, v_3, v_4, t)$. Algorithm 6.1 computes this solution if we initially choose the cut $(S, T)$ that is defined by $S = \{s, v_3\}$. $\triangleleft$

**Example 6.4.** Choosing a minimum $s$-$t$-cut $(S, T)$ in Line 2 of Algorithm 6.1 does not result in an optimal solution in general. Figure 6.2 shows a graph, for which the unique minimum $s$-$t$-cut $(\{s\}, V \setminus \{s\})$ contains the two arcs $su_1$ and $su_2$. If we use this cut as initial $s$-$t$-cut in Algorithm 6.1, we obtain six forbidden pairs. However, starting with the cut $(V \setminus \{t\}, \{t\})$ results in an optimal solution with only three forbidden pairs. $\triangleleft$

**Example 6.5.** In Lines 5 and 8 of Algorithm 6.1 it is in general not sufficient to consider the cuts $(S_u, T_u)$ and $(S_v, T_v)$ only in $G[S]$ and $G[T]$, respectively. To see this, consider the graph from Figure 6.3 and the cut with $S = \{s, u_1, u_2\}$. Considering the minimal $s$-$u$- and $v$-$t$-cuts only in $G[S]$ and $G[T]$, respectively, results in the set of forbidden pairs $\mathcal{A} = \{\{u_1v_1, v_1t\}, \{su_2, u_2v_2\}\}$. However, the path $P = (s, u_1, v_1, u_2, v_2, t)$ is not covered by a forbidden pair in this case. $\triangleleft$

The previous example shows that we cannot simultaneously restrict the minimal cuts in Algorithm 6.1 to the graphs $G[S]$ and $G[T]$. Nonetheless, we can restrict one of the cuts as the following lemma states.

**Lemma 6.6.** In Algorithm 6.1 we can consider the cuts $(S_u, T_u)$ in $G[S]$ if we consider the cuts $(S_v, T_v)$ in $G$. Conversely, we can consider the cuts $(S_v, T_v)$ in $G[T]$ if we consider the cuts $(S_u, T_u)$ in $G$.

*Proof.* We prove that we can determine the cuts $(S_u, T_u)$ in $G[S]$ instead of $G$. The other case is analogous. So, assume we consider the cuts $(S_u, T_u)$ in $G[S]$ and the cuts $(S_v, T_v)$ in $G$. Let $P$ be some $s$-$t$-path and let $a = (u, v)$ be the first arc on $P$ in $\delta^{\text{out}}(S)$. If $(S_a, T_a)$ is the cut $(S_v, T_v)$ in $G$, then the path $P$ has an arc $a' \neq a$ which is contained in $(S_v, T_v)$. Thus, the path $P$ contains the pair $\{a, a'\} \in \mathcal{A}$. Now, consider the case that $(S_a, T_a)$ is the cut $(S_u, T_u)$ in $G[S]$. Since $a$ is the first arc of $P$ not contained in $S$, all arcs prior to $a$ in $P$ are contained in $S$, that is, the path $P$ up to the arc $a$ is contained in $G[S]$. Hence, there also is an edge $a' \neq a$ on $P$ which is contained in $(S_u, T_u)$. Thus, the path $P$ contains the pair $\{a, a'\} \in \mathcal{A}$. $\square$

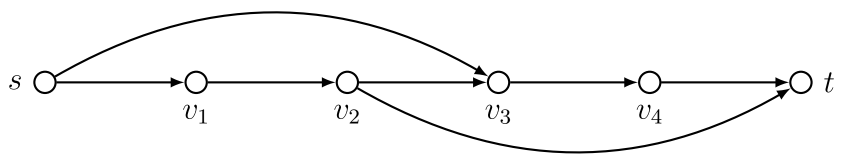


Figure 6.1: The instance from Example 6.3 in which both pairs of the optimal solution $\mathcal{A} = \{\{sv_1, v_1v_2\}, \{v_3v_4, v_4t\}\}$ are on the path $P = (s, v_1, v_2, v_3, v_4, t)$.

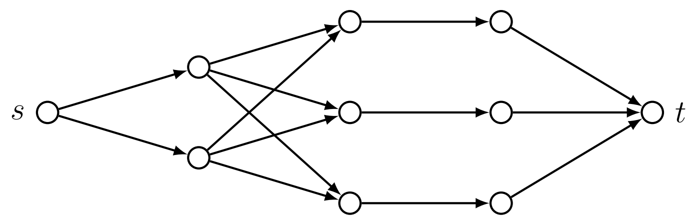


Figure 6.2: A graph for Example 6.4 where choosing a minimum $s$-$t$-cut as initial cut in Algorithm 6.1 does not result in an optimal solution.

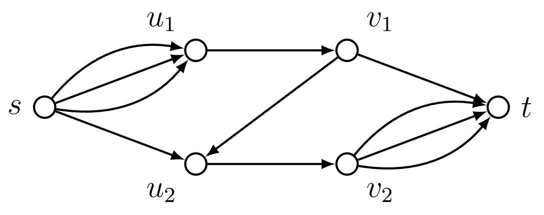
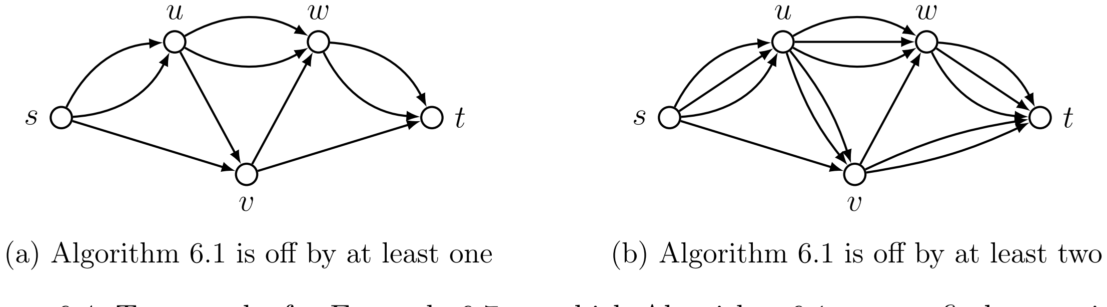


Figure 6.3: An instance for Example 6.5. Defining $(S,T)$ by $S = \{s, u_1, u_2\}$ and restricting the minimum $s$-$u$- and $v$-$t$-cuts to $G[S]$ and $G[T]$, respectively, results in forbidden pairs not covering the path $P = (s, u_1, v_1, u_2, v_2, t)$.



$u$
$w$
$s$
$t$
$v$
(a) Algorithm 6.1 is off by at least one
$u$
$w$
$s$
$t$
$v$
(b) Algorithm 6.1 is off by at least two

Figure 6.4: Two graphs for Example 6.7 on which Algorithm 6.1 cannot find an optimal solution, independent on the minimum cut that is chosen.

| $S$ | $\delta^{\text{out}}(S)$ | $G[S], G$ | | $G, G[T]$ | |
|---|---|---|---|---|---|
| | | $\lvert\delta^{\text{out}}(S_a)\rvert$ | $\lvert\mathcal{A}\rvert$ | $\lvert\delta^{\text{out}}(S_a)\rvert$ | $\lvert\mathcal{A}\rvert$ |
| | $(su)^1$ | 3 | | 3 | |
| $\{s\}$ | $(su)^2$ | 3 | 8 | 3 | 8 |
| | $sv$ | 2 | | 2 | |
| | $sv$ | 2 | | 2 | |
| $\{s, u\}$ | $uv$ | 2 | 8 | 2 | 8 |
| | $(uw)^1$ | 2 | | 2 | |
| | $(uw)^2$ | 2 | | 2 | |
| | $(su)^1$ | 3 | | 2 | |
| $\{s, v\}$ | $(su)^2$ | 3 | 8 | 2 | 8 |
| | $vw$ | 1 | | 2 | |
| | $vt$ | 1 | | 2 | |
| | $(su)^1$ | 3 | | 1 | |
| | $(su)^2$ | 3 | | 1 | |
| $\{s, w\}$ | $sv$ | 2 | 8 | 1 | 9 |
| | $(wt)^1$ | 0 | | 3 | |
| | $(wt)^2$ | 0 | | 3 | |
| | $(uw)^1$ | 2 | | 2 | |
| $\{s, u, v\}$ | $(uw)^2$ | 2 | 8 | 2 | 8 |
| | $vw$ | 2 | | 2 | |
| | $vt$ | 2 | | 2 | |
| | $sv$ | 2 | | 1 | |
| $\{s, u, w\}$ | $uv$ | 2 | 8 | 1 | 8 |
| | $(wt)^1$ | 2 | | 3 | |
| | $(wt)^2$ | 2 | | 3 | |
| | $(su)^1$ | 3 | | 0 | |
| | $(su)^2$ | 3 | | 0 | |
| $\{s, v, w\}$ | $vt$ | 1 | 9 | 2 | 8 |
| | $(wt)^1$ | 1 | | 3 | |
| | $(wt)^2$ | 1 | | 3 | |
| | $vt$ | 2 | | 2 | |
| $\{s, u, v, w\}$ | $(wt)^1$ | 3 | 8 | 3 | 8 |
| | $(wt)^2$ | 3 | | 3 | |

Table 6.1: All possible $s$-$t$-cuts and the corresponding cardinalities of the resulting set $\mathcal{A}$ of forbidden pairs generated by Algorithm 6.1. A table for Example 6.7.

However, also with the improvement of Lemma 6.6 there are instances where Algorithm 6.1 cannot yield an optimal solution, no matter which initial $s$-$t$-cut is chosen. One such instance is presented in the following example.

**Example 6.7.** Consider the graph from Figure 6.4a. We distinguish parallel arcs with superscript indices we write, for example, $(uw)^1$ and $(uw)^2$ for the arcs from $u$ to $w$. The following 7 pairs are a feasible solution of SFP:

$$\begin{array}{ll}
p_1 = \{(su)^1, (wt)^1\} & p_5 = \{sv, vw\} \\
p_2 = \{(su)^1, (wt)^2\} & p_6 = \{sv, vt\} \\
p_3 = \{(su)^2, (wt)^1\} & p_7 = \{uv, vt\} \\
p_4 = \{(su)^2, (wt)^2\} &
\end{array}$$

A path with $sv$ as first arc has either $vw$ or $vt$ as second arc. Thus, it contains either the pair $p_5$ or $p_6$. A path starting with $(su)^1$ (the case $(su)^2$ is analogous) either uses $uv$ and $vt$, thus containing $p_7$, or visits the vertex $w$. In the latter case it uses either $(wt)^1$ or $(wt)^2$ and, hence, contains $p_1$ or $p_2$, respectively.

The solutions of Algorithm 6.1 with the improvement of Lemma 6.6 for all possible $s$-$t$-cuts $(S, T)$ are enumerated in Table 6.1. For any $s$-$t$-cut $(S, T)$ (in the table given by $S$) we get a set $\mathcal{A}$ of separating pairs of cardinality at least 8. Moreover, this is the case no matter if we restrict the cut $(S_u, T_u)$ to $G[S]$ or the cut $(S_v, T_v)$ to $G[T]$.

One can also extend this example such that we have an optimality gap of two. Although 16 forbidden pairs suffice to separate $s$ and $t$ in the graph depicted in Figure 6.4b, Algorithm 6.1 always yields a solution with at least 18 pairs. ◁

## 6.2 $\Sigma_2^p$-Completeness

In this section, we prove that SFP is $\Sigma_2^p$-complete. More precisely, this section is about the proof of the following theorem.

**Theorem 6.8.** SFP is $\Sigma_2^p$-complete, even on acyclic graphs.

First, in Lemma 6.9, we prove the simple part, namely that SFP is contained in $\Sigma_2^p$. Afterwards, for the hardness of the problem, we present a reduction from the $\Sigma_2^p$-complete quantified satisfiability problem $\Sigma_2$SAT. Since this reduction is a bit more complicated, we start by first formulating $\Sigma_2$SAT and by sketching the idea of the proof.

**Lemma 6.9.** SFP is contained in $\boldsymbol{\Sigma}_2^p$.

*Proof.* To prove the claim, it suffices to show that SFP can be solved by a nondeterministic Turing machine that has access to an oracle for an **NP**-complete problem [see AB09, Chapter 5.5]. In the proof we use the notation of certificates that correspond to the nondeterministic choices of the Turing machine [see AB09, Theorem 2.6], and we assume the oracle to answer PAFP.

Given an instance of SFP, we use a separating set $\mathcal{A}$ of forbidden pairs as certificate. With the help of the PAFP-oracle, we can check in constant time whether this is indeed a separating set. Additionally, we can determine the size $k = |\mathcal{A}|$ in time that is linear in $|\mathcal{A}|$.

For a satisfiable SFP instance we can choose a separating set of size $k$ as certificate. Given an SFP instance that is not satisfiable and a certificate $\mathcal{A}$, we can either detect that the size of $\mathcal{A}$ does not equal $k$ or that $\mathcal{A}$ does not separate $s$ and $t$. □

## The Problem $\Sigma_2$SAT

An instance of $\boldsymbol{\Sigma}_2$SAT is given by a quantified Boolean formula $\varphi(x, y)$ depending on two types of variables. The question is, whether an assignment of the $x$-variables exists such that $\varphi(x, y)$ is `true` for every assignment of the $y$-variables. This problem, sometimes also denoted by QSAT$_2$, is a standard $\boldsymbol{\Sigma}_2^p$-complete problem, see [Pap94, Theorem 17.10] or [Haa19, Section 2.2.1]. We first introduce some notation that we use in order to deal with this problem.

**Notation 6.10.** The quantified Boolean formula $\varphi = \varphi(x, y)$ depends on $n_x$ many $x$-variables $X = \{x_1, \ldots, x_{n_x}\}$ and on $n_y$ many $y$-variables $Y = \{y_1, \ldots, y_{n_y}\}$ whose union we denote by $Z = X \cup Y$. A *truth assignment* $T\colon Z \to \{0, 1\}$ assigns a Boolean value to every variable. If we are only interested in the assignments of $x$- or $y$-variables, we write $T_X\colon X \to \{0, 1\}$ as well as $T_Y\colon Y \to \{0, 1\}$ and identify $T$ with $(T_X, T_Y)$, where $T_X = T|_X$ and $T_Y = T|_Y$.

We say that the instance $\varphi$ is *satisfiable* if an $x$-variable assignment $T_X$ exists such that $\varphi$ evaluates to `true` for every $y$-variable assignment $T_Y$.

## Outline of the $\Sigma_2^p$-Hardness Proof

To prove the hardness of SFP, we construct a directed acyclic graph $G$ for such a quantified Boolean formula $\varphi$. For carefully chosen $k \in \mathbb{N}$ we show that a source $s$ and a target $t$ in $G$ can be separated by a set $\mathcal{A}$ of $k$ forbidden pairs if and only if the $\boldsymbol{\Sigma}_2$SAT instance $\varphi$ is satisfiable.

In this graph $G$, most separating pairs are predetermined. Those that are not have essentially two options, which are used to encode assignments of the $x$-variables. This means that an assignment $T_X$ of the $x$-variables corresponds to a selection of forbidden pairs $\mathcal{A}$ and vice versa. An assignment $T_Y$ of the $y$-variables will correspond to $s$-$t$-paths in the graph that contain a pair from $\mathcal{A}$ if and only if the assignment $T = (T_X, T_Y)$ satisfies a clause. From this we conclude that an assignment $T_X$ exists such that $\varphi$ evaluates to `true` for all assignments $T_Y$ if and only if there exists a small set $\mathcal{A}$ such that every $s$-$t$-path contains a pair from $\mathcal{A}$. However, the construction of the graph also generates $s$-$t$-paths that do not correspond to any $y$-variable assignment $T_Y$. To make the argumentation work, we have to enforce that all these paths contain forbidden pairs.

In the following, we start with non-restrictive assumptions about the Boolean formula $\varphi$. Thereafter, we introduce the gadgets and concepts required for the final $\mathbf{\Sigma}_2^p$-hardness proof.

## Assumptions and Assignments

Without loss of generality we may assume that the Boolean formula $\varphi$ is given in 3-DNF, that is, in disjunctive normal form where each clause contains exactly three literals [see Haa19, Section 2.2.1]. Hence, we can write $\varphi = C_1 \vee \cdots \vee C_m$ as a disjunction of $m$ clauses where each clause is the conjunction of three literals.

**Assumption 6.11.** The Boolean formula $\varphi$ is given in 3-DNF.

Let us consider a clause consisting entirely of $x$-variables. If it contains a variable $x_i$ and its negation $\overline{x_i}$, the clause can never be fulfilled and we can remove it. Otherwise, we can satisfy this clause (and with it the entire formula $\varphi$) solely by an appropriate $x$-variable assignment. Hence, we may also assume that every clause contains at least one $y$-variable.

**Assumption 6.12.** No clause of $\varphi$ consists entirely of $x$-variables.

Our last assumption is that no variable is contained in a single clause only. This can be guaranteed, for example, by duplicating all clauses.

**Assumption 6.13.** Every variable is contained in at least two clauses of $\varphi$.

Assumptions 6.11 and 6.12 directly imply the following lemma.

**Lemma 6.14.** Every clause contains either one, two, or three $y$-variables.

Before we describe the graph construction in detail, we introduce local as well as global $y$-variable assignments and define inconsistencies.

**Notation 6.15.** The Boolean formula $\varphi = C_1 \vee \cdots \vee C_m$ is given as a disjunction of $m$ clauses, where every clause $C_i = \ell_i^1 \wedge \ell_i^2 \wedge \ell_i^3$ is a conjunction of exactly three literals $\ell_i^j \in \{z, \overline{z} : z \in Z\}$. By $Y(C) \subseteq Y$ we denote the set of $y$-variables that occur (negated or not) in a clause $C$. We call an assignment of these variables a *local ($y$-variable) assignment* and denote it by $T_{Y(C)} \colon Y(C) \to \{0,1\}$. In the same spirit, we call $T_Y$ a *global assignment.*

**Definition 6.16.** Local $y$-variable assignments $L = T_{Y(C)}$ and $L' = T_{Y(C')}$ for distinct clauses $C$ and $C'$ are *consistent* if they coincide on $Y(C) \cap Y(C')$. Otherwise, they are *inconsistent* and the pair $I = \{L, L'\}$ is an *inconsistency.*

## Graph Components

We are now ready to start with our graph construction that is based on several gadgets.

**Inconsistency Gadgets** We start with the simplest gadget, the *inconsistency gadget.* They correspond to inconsistencies and their only purpose is to enforce that a minimal separating set $\mathcal{A}$ contains a specific pair of arcs. We use this gadget to ensure that paths not corresponding to a global $y$-variable assignment contain a forbidden pair.

Every inconsistency gadget is a directed acyclic graph as depicted in Figure 6.5. It consists of an $s^I$-$t^I$-path with five arcs where the first, third, and last arc is replaced by two parallel arcs.

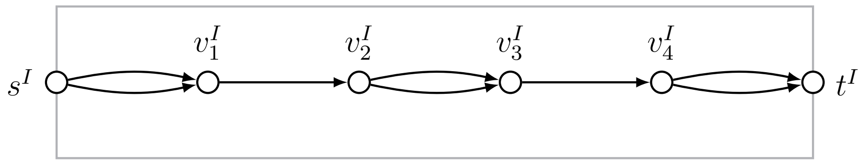


Figure 6.5: An inconsistency gadget corresponding to an inconsistency $I$.

**Lemma 6.17.** The unique optimal solution to separate $s^I$ and $t^I$ in an inconsistency gadget by forbidden pairs is $\mathcal{A}_I = \{\{v_1^I v_2^I, v_3^I v_4^I\}\}$.

*Proof.* $\mathcal{A}_I$ separates $s^I$ and $t^I$ and every separating set needs at least one pair. Thus, every optimal solution consists of a single forbidden pair. To prove the uniqueness, suppose there is an optimal solution whose pair contains one of two parallel arcs. In this case, a path using the other arc does not completely contain this pair, which yields a contradiction. $\square$

**Variable Gadgets** The *variable gadgets* correspond to the $x$-variables in $\varphi$. Their purpose is to reflect a truth assignment $T_X$ of these variables. That is, there should be exactly two optimal sets of forbidden pairs that separate such a gadget: one corresponding to setting the variable to `true` and one for making it `false`. An illustration of such a gadget is given in Figure 6.6. We now describe its construction in more detail. Thereafter, we explain what the two separating sets look like and prove that these are indeed the only two optimal solutions.

Basically, the variable gadget corresponding to a variable $x_i$ consists of two vertices $s^i$ and $t^i$ that are connected by several paths. Similar to the inconsistency gadgets we double some arcs on these paths and we link them in a certain way.

The gadget contains an $s^i$-$t^i$-path for every occurrence of $x_i$ in the formula $\varphi$. More precisely, the $j$-th occurrence corresponds to a path $(s^i, v^i_{j,1}, \dots, v^i_{j,7}, t^i)$ on which we replace the first, fourth, fifth, and last arc by two parallels. Additionally, we add a path $(s^i, v^i_{0,1}, v^i_{0,2}, v^i_{0,3}, v^i_{0,5}, v^i_{0,6}, v^i_{0,7}, t^i)$, which is not associated with any occurrence. On this path we replace the first, fourth, and last arc by two parallel arcs. Furthermore, we introduce the arcs $v^i_{0,3}v^i_{j,4}$ and $v^i_{j,4}v^i_{0,5}$ between these paths.

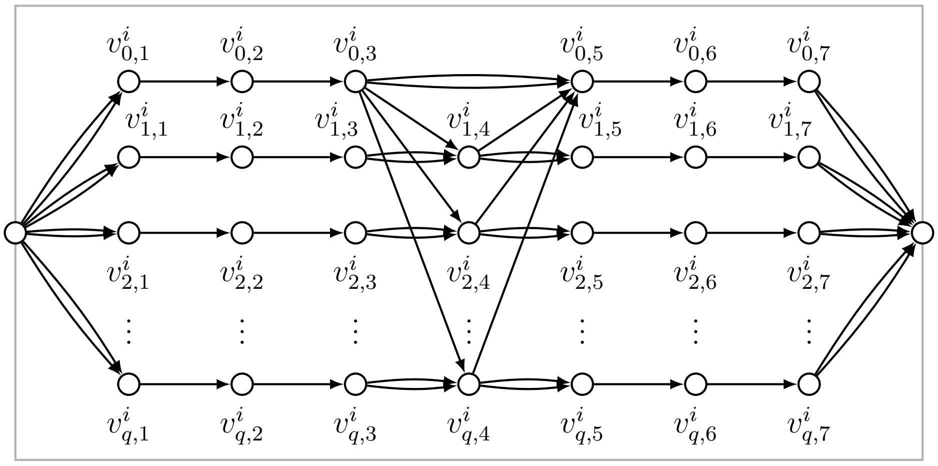


Figure 6.6: A variable gadget corresponding to variable $x_i$. We use $q = q_i$ for the number of occurrences of $x_i$ (including negated literals) in formula $\varphi$.

Let $q_i$ denote the number of occurrences of variable $x_i$ in the formula $\varphi$. As there are, by construction, $q_i + 1$ arc-disjoint $s^i$-$t^i$-paths in this gadget, an optimal set of forbidden pairs separating $s^i$ and $t^i$ must contain at least $q_i + 1$ pairs. Thus, the two separating sets $\mathcal{A}_i = \{\{v^i_{j,1}v^i_{j,2}, v^i_{j,2}v^i_{j,3}\} : j = 0, \dots, q_i\}$ and $\overline{\mathcal{A}_i} = \{\{v^i_{j,5}v^i_{j,6}, v^i_{j,6}v^i_{j,7}\} : j = 0, \dots, q_i\}$ are optimal. The following lemma shows that these are in fact the only two optimal separating sets of forbidden pairs. We identify choosing the separating set $\mathcal{A}_i$ with setting $x_i$ to `true` and choosing $\overline{\mathcal{A}_i}$ with setting $x_i$ to `false`.

**Lemma 6.18.** The sets $\mathcal{A}_i$ and $\overline{\mathcal{A}_i}$ are the only optimal sets of forbidden pairs separating $s^i$ and $t^i$ in the variable gadget corresponding to variable $x_i$.

*Proof.* As argued above, an optimal separating set contains exactly $q_i + 1$ pairs, thus $\mathcal{A}_i$ and $\overline{\mathcal{A}_i}$ are optimal separating sets. It remains to prove the uniqueness.

Similarly to the proof of Lemma 6.17 we can show that no forbidden pair of an optimal solution uses one of two parallel arcs: otherwise, there are still $q_i + 1$ disjoint $s^i$-$t^i$-paths, none of which completely contains this pair. With the same argumentation it follows that none of the arcs $v^i_{0,3}v^i_{j,4}$ and $v^i_{j,4}v^i_{0,5}$ between these paths is contained in a forbidden pair of an optimal solution.

Thus, all forbidden pairs are composed of arcs of the form $v^i_{j,1}v^i_{j,2}$, $v^i_{j,2}v^i_{j,3}$, $v^i_{j,5}v^i_{j,6}$, and $v^i_{j,6}v^i_{j,7}$. For $j \in \{1, \dots, q_i\}$ we consider the four different paths

$$(s^i, v^i_{0,1}, \dots, v^i_{0,7}, t^i), \qquad (s^i, v^i_{0,1}, v^i_{0,2}, v^i_{0,3}, v^i_{j,4}, \dots, v^i_{j,7}, t^i),$$
$$(s^i, v^i_{j,1}, \dots, v^i_{j,7}, t^i), \text{ and } \qquad (s^i, v^i_{j,1}, \dots, v^i_{j,4}, v^i_{0,5}, v^i_{0,6}, v^i_{0,7}, t^i).$$

An optimal solution has to separate these four paths with only two forbidden pairs as there are $q_i - 1$ disjoint paths in the remaining gadget. This, however, is only possible if either the pairs $\{v^i_{0,1}v^i_{0,2}, v^i_{0,2}v^i_{0,3}\}$ and $\{v^i_{j,1}v^i_{j,2}, v^i_{j,2}v^i_{j,3}\}$ or the pairs $\{v^i_{0,5}v^i_{0,6}, v^i_{0,6}v^i_{0,7}\}$ and $\{v^i_{j,5}v^i_{j,6}, v^i_{j,6}v^i_{j,7}\}$ are chosen. Since this holds for all $j \in \{1, \dots, q_i\}$, the claim follows. □

**Formula Gadget** The *formula gadget* consists of clause assignment units, which we describe later, that are ordered in a layered structure. For now it suffices to know that they have one input vertex and one output vertex which we use to connect them. By Lemma 6.14, every clause $C$ of $\varphi$, contains $\ell \in \{1, 2, 3\}$ many $y$-variables. For every of the $2^\ell$ possible local $y$-variable assignments $L = T_{Y(C)}$ for $C$ we introduce one such clause assignment unit. We denote its input vertex by $s^L$ and its output vertex by $t^L$. This yields either two, four, or eight clause assignment units for each clause.

The $i$-th layer of the formula gadget consists of all clause assignment units corresponding to the $i$-th clause of $\varphi$. A source $s^0$ is connected to the input $s^L$ of every clause assignment unit corresponding to a local assignment $L = T_{Y(C_1)}$ of the first clause. In addition, we connect the clause assignment units of successive clauses in the formula gadget by complete bipartite graphs. Finally, we connect every output $t^L$ of a unit corresponding to the last clause $C_m$ with the target $t^0$. The structure of the formula gadget is visualized in Figure 6.7.

Most clause assignment units provide paths from their input to their output vertex. Therefore, $s^0$-$t^0$-paths through the formula gadget pass through exactly one clause assignment unit of every layer. This way, every such path selects a local $y$-variable assignment for every clause. If these are consistent, that is, if every $y$-variable is assigned the same truth value in each clause assignment that contains it, they can be combined

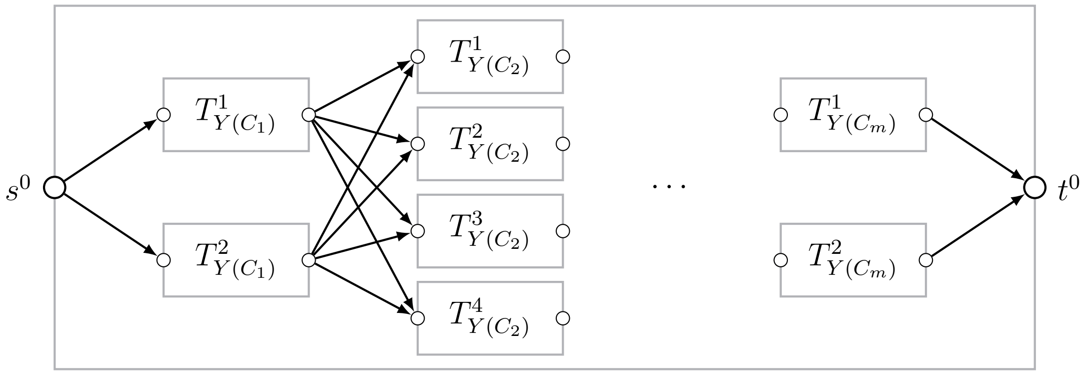


Figure 6.7: The formula gadget for the formula $\varphi = C_1 \vee C_2 \vee \cdots \vee C_m$ in 3-DNF. To distinguish the different possible local assignments of a clause $C$ we enumerate them $T^1_{Y(C)}$, $T^2_{Y(C)}$, and so on.

to a global $y$-variable assignment. The other way around, we can also associate an assignment $T_Y$ with an $s^0$-$t^0$-path which uses in every layer the clause assignment unit corresponding to $T_{Y(C)} = T_Y|_{Y(C)}$ for the respective clause $C$.

Thus, the paths through the formula gadget are linked with the global $y$-variable assignments. Our goal is to ensure that any such path contains a forbidden pair if and only if the associated assignment satisfies the formula $\varphi$ (in conjunction with the $x$-variable assignment). For this, the variable gadgets will play an important role. However, we also have to take those paths into consideration that do not correspond to consistent $y$-variable assignments. In order to ensure that these paths contain forbidden pairs as well, we will make use of the inconsistency gadgets.

**Typification** All the gadgets introduced until now need to be part of $s$-$t$-paths in the final graph. In order to keep mixed paths in check when we finally put these pieces together we need the concept of typification.

To explain the idea of the *typification* concept, we start with a small example. Given are two disjoint graphs $G_1$ and $G_2$. In each graph $G_i$ we want to separate a source $s^i$ and a target $t^i$ by forbidden pairs. However, we want to combine these two graphs to a single graph $G$ without affecting the optimal choice of forbidden pairs, that is, we still only want to select pairs in $G_1$ and $G_2$. Simply adding a source $s$, a target $t$, and connecting these with arcs $ss^1$, $ss^2$, $t^1t$, and $t^2t$ does not suffice as the combined instance can always be separated by the two forbidden pairs $\{ss^1, t^1t\}$ and $\{ss^2, t^2t\}$. But if we know that $p-1$ pairs are sufficient to separate $G_i$ for $i \in \{1,2\}$, we can replace each of the four additional arcs by a bunch of $p$ parallel arcs. In other words: if $k_i$ pairs are sufficient to separate $G_i$, we can choose any $p > \max\{k_1, k_2\}$. Therefore, an optimal solution in the combined instance only uses arcs that are contained within the subgraphs $G_1$ and $G_2$.

If we also add $p+1$ parallel arcs from $s^1$ to $t^2$ as well as from $s^2$ to $t^1$, we have to choose forbidden pairs separating all paths $(s, s^1, t^2, t)$ and all paths $(s, s^2, t^1, t)$. These paths only consist of additional arcs not contained in the original graphs $G_1$ and $G_2$. The unique optimal solution to separate these paths is to choose the $2p^2$ forbidden pairs that combine an arc $ss^1$ with an arc $t^2t$ and an arc $ss^2$ with an arc $t^1t$. Thus, we can separate $G_1$ by $k_1$ forbidden pairs and $G_2$ by $k_2$ forbidden pairs if and only if we can separate $G$ by $2p^2 + k_1 + k_2$ forbidden pairs. This situation is visualized in Figure 6.8.

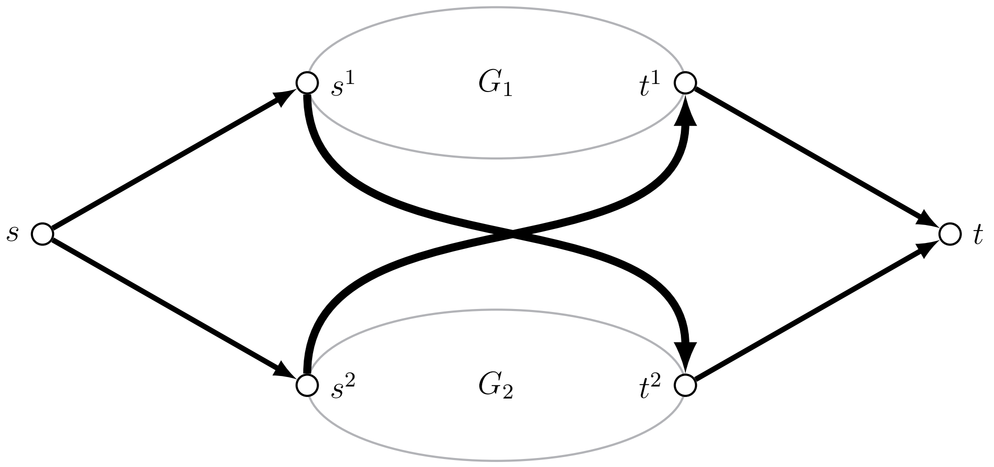


Figure 6.8: An exemplary typification construction. The bold arcs $ss^1$, $ss^2$, $t^1t$, and $t^2t$ represent bunches of $p$ parallel arcs. The even thicker arcs $s^1t^2$ and $s^2t^1$ represent bunches of $p+1$ parallel arcs.

The reason we introduce these additional arcs is because they help us weed out mixed paths: if we allow arcs between $G_1$ and $G_2$ in $G$, then it becomes possible to obtain $s$-$t$-paths containing $s^i$ and $t^j$ for $i \neq j$. By adding the additional "diagonal" arcs $s^1t^2$ and $s^2t^1$ we enforce the choice of all $2p^2$ pairs $\{ss^i, t^jt\}$ for $i \neq j$ and, thus, ensure that these mixed paths are already saturated with at least one pair. This just leaves paths that start with $ss^i$ and end with $t^it$ for all possible indices $i$. We only have to examine whether all paths of these two *types* contain a forbidden pair or not. Note that this does include paths that are not solely part of a subgraph $G_i$, as they can leave and return, but it does reduce the potential paths without a forbidden pair immensely.

This construction can be generalized to more than only two types. For $q$ subgraphs $G_1, \dots, G_q$ with sources $s^i$ and targets $t^i$, $i \in \{1, \dots, q\}$, we can add $p$ parallel arcs from $s$ to every source $s^i$ and from every target $t^i$ to $t$. Additionally, we add $p+1$ parallel arcs $s^it^j$ for all $i, j \in \{1, \dots, q\}$ with $i \neq j$. Every optimal solution has to use the $p^2$ forbidden pairs of arcs $\{ss^i, t^jt\}$ of different types $i \neq j$. Thus, every optimal solution has $p^2q(q-1)$ forbidden pairs and, additionally, the pairs required to separate all paths of the $q$ different types (all paths using $ss^i$ and $t^it$ for some $i$). We intend to use this to give all inconsistency gadgets, all variable gadgets, as well as the formula gadget their own type.

**Clause Assignment Units and Graph Construction** To construct the graph corresponding to formula $\varphi$ we use the formula gadget, a variable gadget for every $x$-variable, and several inconsistency gadgets. More precisely, for every pair of clause assignment units (within the formula gadget) that corresponds to incompatible assignments we introduce one such inconsistency gadget. All these gadgets are combined into the graph $G$ as explained in the typification section.

Let us describe the graph construction in detail. That is, we finally have to specify what the clause assignment units look like and how these are connected to the other gadgets. Recall that the formula gadget contains a clause assignment unit for every clause $C$ and every possible assignment $L = T_{Y(C)}$ of Boolean values to the $y$-variables contained in $C$. As already stated in the formula gadget section, these are $2^\ell$ clause assignment units for a clause with $\ell$ many $y$-variables.

A *clause assignment unit* corresponding to a $y$-variable assignment $L$ of a clause $C$ contains exactly three vertices: $s^L$, $v^L$, and $t^L$. Note that we use $s^L$ and $t^L$ in order to connect the clause assignment units in the formula gadget as described above. The vertices $v^L$ and $t^L$ are connected by an arc $v^L t^L$ if and only if $C$ contains at least one $y$-literal that evaluates to `false` with the $y$-variable assignment $L$. This arc is missing in exactly one clause assignment unit corresponding to a clause $C$ as there is only exactly one assignment $T_{Y(C)}$ that satisfies all $y$-literals in $C$.

These are all components within a clause assignment unit. In particular, the clause assignment units are not connected and, thus, neither is the formula gadget. The following modifications only add some arcs between different gadgets. These are illustrated by dashed arcs in Figures 6.9 to 6.11.

In addition to the (potentially non-existing) arc $v^L t^L$, we add another path from $v^L$ to $t^L$ for every $x$-literal contained in $C$. The path of a literal corresponding to variable $x_i$ passes through the variable gadget of $x_i$. If the occurrence of $x_i$ in $C$ is the $j$-th occurrence in $\varphi$ in total, this path uses either the arcs $v^i_{j,1} v^i_{j,2}$ and $v^i_{j,2} v^i_{j,3}$ (if $C$ contains the literal $x_i$) or the arcs $v^i_{j,5} v^i_{j,6}$ and $v^i_{j,6} v^i_{j,7}$ (if $C$ contains the literal $\overline{x}_i$). In the former case we add the inter-gadget arcs $v^L v^i_{j,1}$ and $v^i_{j,3} t^L$ and in the latter case we add $v^L v^i_{j,5}$ as well as $v^i_{j,7} t^L$. These connecting arcs are indicated by dashed orange arcs in Figures 6.9 and 6.10.

## Magnitudes and Parameters

Recall that we denote the number of clauses of $\varphi = C_1 \vee \cdots \vee C_m$ by $m$ and the number of $x$- and $y$-variables by $n_x$ and $n_y$, respectively (compare Notations 6.10 and 6.15). Also as before, let $q_i$ denote the number of occurrences of the $i$-th $x$-variable $x_i$ in the formula $\varphi$. Additionally, we denote the number of inconsistencies by $n_I$.

The graph of the corresponding SFP instance consists of one formula gadget, $n_x$ variable gadgets, and $n_I$ inconsistency gadgets. The formula gadget consists of at most eight clause

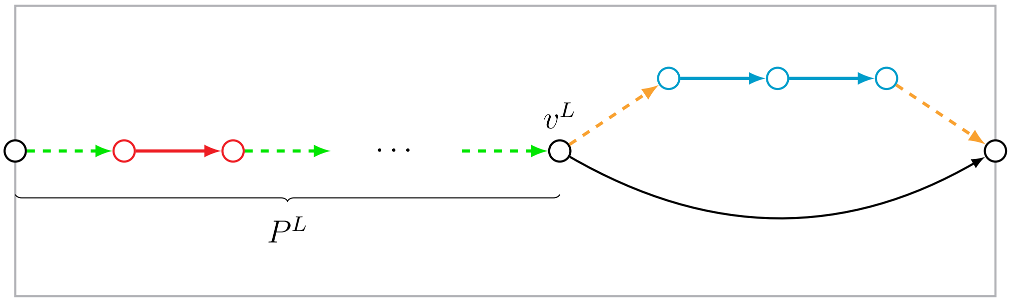


Figure 6.9: A clause assignment unit corresponding to a local $y$-variable assignment $L$ for clause $C$ containing a single $x$-variable. The assignment $L$ does not fulfill all $y$-literals of $C$ as the arc $v^L t^L$ is present. The colored, solid arcs are contained in other gadgets and the dashed arcs connect these, see also Figures 6.10 and 6.11.

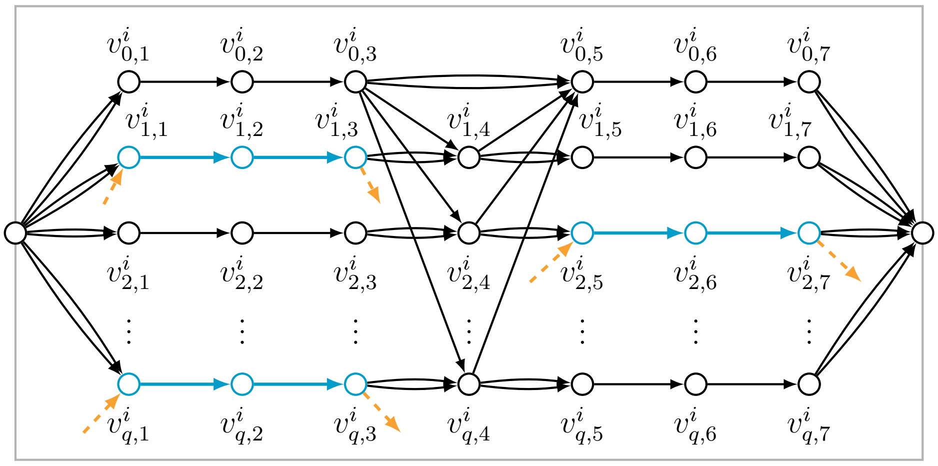

$s^i$
$t^i$
$v^i_{0,1}$ $v^i_{0,2}$ $v^i_{0,3}$ $v^i_{0,5}$ $v^i_{0,6}$ $v^i_{0,7}$
$v^i_{1,1}$ $v^i_{1,2}$ $v^i_{1,3}$ $v^i_{1,4}$ $v^i_{1,5}$ $v^i_{1,6}$ $v^i_{1,7}$
$v^i_{2,1}$ $v^i_{2,2}$ $v^i_{2,3}$ $v^i_{2,4}$ $v^i_{2,5}$ $v^i_{2,6}$ $v^i_{2,7}$
$v^i_{q,1}$ $v^i_{q,2}$ $v^i_{q,3}$ $v^i_{q,4}$ $v^i_{q,5}$ $v^i_{q,6}$ $v^i_{q,7}$

Figure 6.10: A variable gadget (as in Figure 6.6) with the connections to clause assignment units. The blue arcs and the dashed orange arcs correspond to these from Figure 6.9. In this example, the first and last occurrence of the corresponding $x$-variable occurs non-negated and the second occurrence is negated.

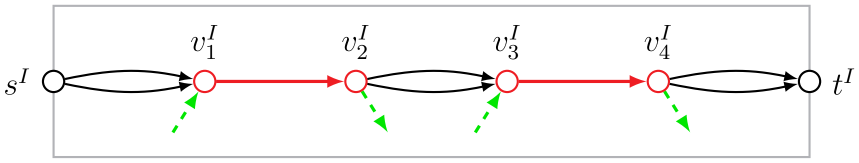


Figure 6.11: An inconsistency gadget (as in Figure 6.5) with the connections to clause assignment units or other inconsistency gadgets. The red arcs and the green dashed arcs correspond to these from Figure 6.9.

assignment units per clause. Hence, we have $\mathcal{O}(m)$ clause assignment units. Moreover, since we have at most one inconsistency gadget for every pair of clause assignment units, it holds that $n_I \in \mathcal{O}(m^2)$.

As shown in Lemmas 6.17 and 6.18 we need one forbidden pair to separate every inconsistency gadget and $q_i + 1$ forbidden pairs to separate the variable gadget for $x_i$. Since the formula gadget itself is not connected, we do not need additional forbidden pairs to separate it. Thus, for the typification framework, we choose

$$p = \max_{i=1,\ldots,n_x} q_i + 2 \tag{6.1}$$

and add $p$ parallel arcs from a source $s$ to all input vertices of variable and inconsistency gadgets as well as to the formula gadget. That is, we add all parallels of the form $ss^i$, $ss^I$, and $ss^0$. Analogously, we add $p$ parallel arcs from every such output vertex to a target $t$ resulting in parallels $t^it$, $t^It$, and $t^0t$. Furthermore, we add $p + 1$ parallel arcs from every input vertex of such a gadget to the output vertices of all other gadgets. In total, we introduce

$$2p(n_x + n_I + 1) + (p + 1)(n_x + n_I + 1)(n_x + n_I) \in \mathcal{O}\left(m^5\right)$$

arcs for the typification. The asymptotic complexity $\mathcal{O}(m^5)$ follows since $n_I \in \mathcal{O}(m^2)$ and since both, $n_x$ and $p$, are bounded by the number $3 \cdot m$ of literals in $\varphi$. As explained in the typification section we need

$$k_0 = p^2(n_x + n_I + 1)(n_x + n_I) \tag{6.2}$$

pairs to separate all paths that only consist of arcs introduced for typification. We show in the analysis section below that we can separate the graph $G$ by

$$k = k_0 + n_I + \sum_{i=1}^{n_x}(q_i + 1) \tag{6.3}$$

forbidden pairs if and only if the quantified Boolean formula $\varphi$ has an $x$-variable assignment such that $\varphi$ evaluates to `true` for every $y$-variable assignment.

## Analysis

So far, given a quantified Boolean formula $\varphi$, we have constructed an SFP instance $G$ with source $s$ and target $t$ and specified the number $k$ of forbidden pairs. In the following, we use this to give a proof for Theorem 6.8, which we divide into a few lemmas. First, in Lemmas 6.19 and 6.20 we prove $G$ to acyclic and that we can construct this graph in polynomial time. Thereafter, we show in Lemma 6.21 that $k$ pairs are required to separate $s$ and $t$. Finally, in Lemmas 6.22 and 6.23 we prove that $k$ pairs actually suffice if and only if $\varphi$ is satisfiable.

**Lemma 6.19.** The graph $G$ that is constructed as described above is acyclic.

*Proof.* As a graph is acyclic if and only if it exists a topological ordering, we prove the claim by specifying such a topological ordering for $G$. However, we do not explicitly map every vertex to a natural number. Instead, we describe a procedure how to obtain the order of the vertices. The reason is that we have to insert some vertices in between others multiple times. This would make a formal definition of this mapping quite technical.

In a first step, we enumerate all vertices in the formula gadget together with the interior vertices from inconsistency gadgets. Here, the "interior vertices" of an inconsistency gadget for an inconsistency $I$ are the vertices $v_1^I, \dots, v_4^I$. Note that each arc from $v_1^I v_2^I$ and $v_3^I v_4^I$ is contained in an $s^L$-$v^L$-path $P^L$ of some clause assignment unit. We start to enumerate the vertices in clause assignment units corresponding to the first clause $C_1$. There, we first enumerate the paths $P^L$ of assignments $L$ for $C_1$ followed by the output vertices $t^L$ of the corresponding gadgets. Afterwards, we proceed in the same way with the subsequent clauses. This procedure is visualized in Figure 6.12.

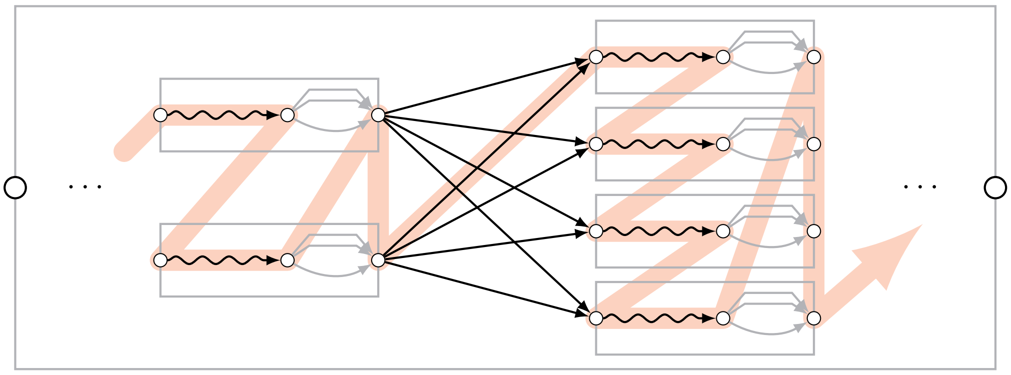


Figure 6.12: A schematic representation how to enumerate the vertices in a formula gadget for a topological ordering. The $s^L$-$v^L$-paths within the clause assignment units are drawn as wavy lines. The path and lines that might connect $v^L$ and $t^L$ are only indicated. This visualizes the first step in the proof of Lemma 6.19.

By enumerating the formula gadget that way, for $i < j$ every vertex corresponding to a clause $C_i$ gets a lower number than every vertex corresponding to clause $C_j$. This holds in particular for the vertices on the $s^L$-$v^L$-paths $P^L$ in the clause assignment units. For every inconsistency $I = \{L = T_{Y(C_i)}, L' = T_{Y(C_j)}\}$ with $i < j$, the path $P^L$ uses the arc $v_1^I v_2^I$ and the path $P^{L'}$ uses the arc $v_3^I v_4^I$ within the corresponding inconsistency gadget. Thus, the partial topological ordering defined up to this point is not only consistent with all arcs of the formula gadget and the arcs in between formula and inconsistency gadgets, but also within all these inconsistency gadgets.

It remains to prove that we can extend this partial ordering to the variable gadgets and the missing in- and output vertices. The latter are, however, no problem as we can put all input vertices at the start and all output vertices at the end, directly after $s$ or before $t$ (except for the in- and outputs of clause assignment units that already are assigned a number in the first step).

Thus, in a second step, we have to assign numbers to the vertices of the variable gadgets. Such a variable gadget corresponding to a variable $x_i$ consists of $q_i + 1$ many $s^i$-$t^i$-paths. With the exception of the additional path, every path corresponds to one occurrence of this variable. Let us consider the $j$-th occurrence and let $C$ be the corresponding clause. Depending on whether $x_i$ occurs negated or not, we have restrictions either for the values of $v^i_{j,5}$ and $v^i_{j,7}$ or for the values of $v^i_{j,1}$ and $v^i_{j,3}$, respectively (as those have arcs to vertices in clause assignment units that are already assigned a number). In particular, we only have restrictions on the "left half" or on the "right half" of the path but not on both. In the case the $j$-th occurrence is not negated, we assign the vertices $v^i_{j,1}, \dots, v^i_{j,3}$ increasing values that we insert in between the highest number of a vertex $v^L$ and the lowest number of a vertex $t^L$ in the topological ordering for every assignment $L$ of clause $C$. Note that we have enumerated these vertices in the first phase, such that the highest number of a vertex $v^L$ is in fact smaller than the lowest number of a vertex $t^L$ for an assignment $L$ of clause $C$.

Because all paths within the variable gadget are only connected to the additional path $(s^i, v^i_{0,1}, v^i_{0,2}, v^i_{0,3}, v^i_{0,5}, v^i_{0,6}, v^i_{0,7}, t^i)$, we can extend the partial topological ordering within every variable gadget. Therefore, we can assign $v^i_{0,1}$, $v^i_{0,2}$, and $v^i_{0,3}$ values that are smaller and $v^i_{0,5}$, $v^i_{0,6}$, and $v^i_{0,7}$ values that are larger than any values of vertices within the variable gadget. Thereafter, we can insert the "missing half" of paths accordingly. □

**Lemma 6.20.** The graph $G$ corresponding to the formula $\varphi$ is of polynomial size and it can be constructed in polynomial time, both with respect to the size of $\varphi$.

*Proof.* For a given instance $\varphi = C_1 \vee \dots \vee C_m$ with $n_x$ many $x$-variables, let $G$ be the graph as described in this section. Its size is polynomial in the size of $\varphi$ as it contains $\mathcal{O}(m)$ clause assignment units, $n_x$ variable gadgets, and $\mathcal{O}(m^2)$ inconsistency gadgets. The size of the clause assignment and inconsistency gadgets is constant and the size of a variable gadget is linear in the number of occurrences of the corresponding $x$-variable. We add $\mathcal{O}(m^5)$ arcs for the typification and to see that also only polynomially many arcs connect different gadgets we can associate these to at least one of the two corresponding gadgets. Every inconsistency gadget is connected by exactly four inter-gadget arcs and every clause assignment unit is connected by either two, four, or six inter-gadget arcs. All the arcs connecting a variable gadget to other gadgets have the other endpoint in a clause assignment unit and are thus already considered. Hence, the amount of inter-gadget arcs is polynomially bounded. Moreover, we can also construct the graph $G$ from the formula $\varphi$ in polynomial time. □

**Lemma 6.21.** At least $k$ forbidden pairs are required to separate $s$ and $t$ in $G$, where $k$ is defined as in Equation (6.3) on Page 68.

*Proof.* This follows from the typification construction. Every set of forbidden pairs $\mathcal{A}$ has to contain

- ▷ the $k_0$ pairs to separate all paths that consist only of typification arcs,
- ▷ the $n_I$ pairs to separate all inconsistency gadgets (see Lemma 6.17), and
- ▷ for every $x_i$ either $\mathcal{A}_i$ or $\overline{\mathcal{A}}_i$ (see Lemma 6.18). □

**Lemma 6.22.** If the $\mathbf{\Sigma}_2$SAT instance $\varphi$ is satisfiable, we can separate $s$ and $t$ in $G$ by $k$ forbidden pairs, where $k$ is defined as in Equation (6.3) on Page 68.

*Proof.* If $\varphi$ is satisfiable, there is an $x$-variable assignment $T_X$ such that $\varphi$ evaluates to `true` no matter which values are assigned to the $y$-variables. We define a set of forbidden pairs depending on $T_X$ as follows.

First, it contains the $k_0$ forbidden pairs to separate all paths only consisting of typification arcs. Second, it contains the $n_I$ pairs that separate all inconsistency gadgets, compare Lemma 6.17. Finally, we choose a separating set for every $x$-variable $x_i$. If $T_X(x_i) = 1$, we use the separating set $\mathcal{A}_i$. Otherwise, we use $\overline{\mathcal{A}_i}$. See Lemma 6.18 for more information on these two sets.

By Equation (6.3) and Lemmas 6.17 and 6.18 we have chosen $k$ forbidden pairs. Moreover, by the typification construction, these pairs separate all paths that do not use any gadget and those whose first and last gadgets are not the same.

It remains to prove that every path that enters a gadget via a direct arc from $s$ and leaves this gadget via a direct arc to $t$ completely contains at least one forbidden pair. We consider the different gadgets.

First, consider an inconsistency $I$ and the corresponding inconsistency gadget. Every path entering this gadget via $ss^I$ must also use the arc $v_1^I v_2^I$. Analogous, every path leaving this gadget via $t^I t$ must also use the arc $v_3^I v_4^I$. Thus, every path entering and leaving this gadget via input $s^I$ and output $t^I$ contains the forbidden pair $\{v_1^I v_2^I, v_3^I v_4^I\}$ that we have chosen.

Next, consider the variable gadget for a variable $x_i$. Similarly to the inconsistency gadget, a path entering the gadget via $s^i$ can leave the gadget at the earliest at some vertex $v_{j,3}^i$ and, thus, it has to contain the pair $\{v_{j,1}^i v_{j,2}^i, v_{j,2}^i v_{j,3}^i\}$. Analogous, a path leaving the gadget via $t^i$ must enter the gadget at the latest at some vertex $v_{j',5}^i$ and, thus, it has to contain the pair $\{v_{j',5}^i v_{j',6}^i, v_{j',6}^i v_{j',7}^i\}$. At least one of these two pairs is contained in the set of forbidden pairs we have chosen.

Finally, let us consider the formula gadget and let $P$ be a path that enters the gadget via $s^0$ and leaves it finally via $t^0$. The path $P$ passes through multiple inconsistency gadgets. If it uses more than one arc from one of them, it directly contains the forbidden pair chosen in this inconsistency gadget. Thus, we can assume that $P$ uses at most one arc from every inconsistency gadget.

If the path $P$ leaves some clause assignment unit for a clause $C$ via an arc to a variable gadget, it has to leave this variable gadget via an arc to the vertex $t^L$ of a clause assignment unit that also corresponds to clause $C$. Thus, for every clause, this path enters exactly one clause assignment unit via its input $s^L$ and uses the $s^L$-$v^L$-path $P^L$ therein. Consequently, if $P$ passes through clause assignment units corresponding to inconsistent assignments $L$ and $L'$, it contains the forbidden pair contained in the inconsistency gadget for $I = \{L, L'\}$.

Therefore, we can assume that $P$ enters only clause assignment units corresponding to consistent assignments. This allows us to define a global $y$-variable assignment $T_Y$ by combining the local clause assignments related to the clause assignment units that $P$ enters via $s^L$. As $\varphi$ is satisfiable and $T_X$ is chosen appropriately we have that $\varphi$ evaluates to `true` with $T = (T_X, T_Y)$. In particular, there is at least one clause $C$ that is fulfilled. Let us consider the clause assignment unit associated to $C$ that $P$ enters via $s^L$. As this clause is fulfilled, all $y$-literals are `true` and, thus, the arc $v^L t^L$ is not present. Hence, the path $P$ has to pass through an $x$-variable gadget. However, as also this $x$-literal in $C$ is `true`, by the construction of the graph and the choice of the forbidden pairs, $P$ has to use a forbidden pair in this variable gadget.

In all possible cases, the path $P$ contains a forbidden pair. Thus, $s$ and $t$ can be separated in $G$ by $k$ forbidden pairs. □

**Lemma 6.23.** If the $\mathbf{\Sigma}_2$SAT instance $\varphi$ is not satisfiable, we cannot separate $s$ and $t$ in $G$ by $k$ forbidden pairs, where $k$ is defined as in Equation (6.3) on Page 68.

*Proof.* Toward a contradiction, suppose that we can separate $s$ and $t$ in $G$ by $k$ forbidden pairs.

By Lemma 6.21 we need at least $k$ forbidden pairs. By the typification construction and by Lemmas 6.17 and 6.18 we have to choose the forbidden pairs from $\mathcal{A}_i$ or $\overline{\mathcal{A}_i}$ in a variable gadget corresponding to variable $x_i$.

We define an $x$-variable assignment $T_X$ based on this set of forbidden pairs. A variable $x_i$ is set to `true` if we have chosen $\mathcal{A}_i$ to separate its variable gadget. Otherwise, if we have chosen $\overline{\mathcal{A}_i}$, we set $x_i$ to `false`.

As $\varphi$ is not satisfiable, there exists a $y$-variable assignment $T_Y$ such that $\varphi$ evaluates to `false` with $T = (T_X, T_Y)$. This $y$-variable assignment $T_Y$ corresponds to exactly one clause assignment unit $T_{Y(C)} = T_Y|_{Y(C)}$ for every clause $C$. We now construct an $s$-$t$-path in $G$ that does not contain a forbidden pair. This path starts with the arc $ss^0$ and ends with $t^0t$. For every clause $C$ it passes through the clause assignment unit corresponding to $L = T_Y|_{Y(C)}$ where it first uses the $s^L$-$v^L$-path $P^L$. If the clause contains a $y$-literal that is `false`, the path continues along the arc $v^Lt^L$ that is present in this case. Otherwise, there is an $x$-literal that is not fulfilled. In this case, there exist a $v^L$-$t^L$-path through the corresponding variable gadget using two arcs that are not chosen as a forbidden pair (as this literal is `false`).

The path constructed this way does not contain a forbidden pair from a variable gadget. It does not contain a forbidden pair from an inconsistency gadget either as it only uses at most one arc from every inconsistency gadget. This is the case because it only uses consistent assignments for the clauses. And since the path does not contain a forbidden pair used to separate paths consisting of only typification arcs, the path does not contain a forbidden pair at all. This contradicts our initial assumption and finishes the proof. □

## Conclusion

The undisputed highlight of this chapter was the $\Sigma_2^p$-completeness proof for SFP, which is based on a reduction form $\Sigma_2$SAT. Previously, we developed a heuristic for MinSFP. This algorithm is often capable of computing optimal separating pairs of arcs. However, we provided an instance that cannot be solved optimally by applying it.

# Part II

# Fastest Paths with Time Profiles and Waiting

*In this part, we deal with a simplified version of the practical problem in the time-dependent setting: the Fastest Path with Time Profiles and Waiting (*FPTPW*) problem. Given a directed acyclic graph with durations on the arcs and time profiles at the vertices, we are interested in a fastest path from a source to a sink vertex. At specified vertices we are allowed to wait, and the solutions have to comply with related temporal constraints.*

*We develop a solution algorithm for* FPTPW*, which propagates "departure-duration functions" as labels. Proving its correctness and bounding its running time requires profound knowledge about the structure of these functions. As a preliminary step to the analysis of these functions, we focus on the temporal information and provide an algorithm that synchronizes the time profiles between the vertices. Although* FPTPW *is **NP**-complete in general, we develop algorithms that allow solving the problem in polynomial time for certain special cases, in particular if waiting is allowed at every vertex.*

# Chapter 7

# The Fastest Path Problem

*We define the Fastest Path with Time Profiles and Waiting (*FPTPW*) problem and prove its **NP**-completeness. To represent feasible solutions we introduce "valid paths", which also play an important role in solving the problem algorithmically. For this purpose, we lay the foundations by providing operations to modify and combine valid paths. Furthermore, we present a mixed integer linear programming formulation for* FPTPW.

## Assumptions

Throughout this chapter, $G = (V, A)$ always denotes a directed acyclic graph with designated source vertex $v_s$ and target vertex $v_t$.

## 7.1 Setting and Problem Definition

Before we are able to define the Fastest Path with Time Profiles and Waiting (FPTPW) problem in Problem 7.6, we need to introduce time profiles, prepare the setting by defining an appropriate network, and specify what valid solutions to the problem look like. We start with the time profiles.

**Definition 7.1** (Time Profile, Time Window)**.** A *time profile* $T \subseteq \mathbb{R}$ is a finite union of closed intervals. A *time window* in $T$ is an inclusion-wise maximal interval $I \subseteq T$. The (unique) number of distinct time windows in $T$ is denoted by $|T|$ and the set of all time profiles by $\mathcal{T}$.

Next, we bundle all the data required for an instance of our fastest path problem in a network. Anticipating the abbreviation for Problem 7.6 we call this an FPTPW network.

**Definition 7.2** (FPTPW network)**.** An FPTPW *network* $N = (G, v_s, v_t, d, tp, W)$ is a tuple consisting of

- ▷ a directed acyclic graph $G = (V, A)$,
- ▷ a source $v_s$ and a target $v_t$ in $V$,
- ▷ a function $d\colon A \to \mathbb{R}$ assigning a duration to every arc,
- ▷ a function $tp\colon V \to \mathcal{T}$ assigning a time profile to every vertex, and
- ▷ a subset $W \subseteq V$ of vertices at which waiting is allowed.

Since we are not only interested in static paths, we have to specify how these depend on the time, referring to the FPTPW instance. Think of walking along a path as follows: We depart at a vertex $u$ at a certain time and reach its successor $v$ after the duration $d(uv)$ of the arc $uv$ has passed. If the vertex $v$ allows, we may wait there before continuing further along the path. A more detailed intuition of this kind of time dependency (and the reason for using it) is given in Section 3.5.

In order to formally grasp the temporal component of paths, we specify the departure times $\tau$. These implicitly define arrival times $\sigma$ as well as waiting times for all vertices on the path. To neatly express the conditions that these departure times must meet in order to call them *valid*, we first define arrival intervals. For a potential departure time $t$ at a vertex $v$, the corresponding arrival interval contains exactly the times at which we can arrive at $v$ in order to be able to depart as desired at time $t$. We denote the arrival interval for a departure time $t$ at a vertex $v$ by $I_v^\sigma(t)$. Note that the superscript $\sigma$ only indicates that this interval specifies possible arrival times. We give intuition about these arrival intervals after defining them formally.

**Definition 7.3** (Arrival Interval)**.** Let $(G, v_s, v_t, d, tp, W)$ be an FPTPW network with $G = (V, A)$. The *arrival interval* $I_v^\sigma(t)$ at a vertex $v \in V$ for departure $t \in \mathbb{R}$ is defined by

$$I_v^\sigma(t) = \begin{cases} \varnothing & \text{if } t \notin tp(v), \\ \{t\} & \text{if } t \in tp(v) \text{ and } v \notin W, \text{ and} \\ [t_0, t] & \text{if } t \in tp(v) \text{ and } v \in W \end{cases}$$

where $t_0$ in the last case is given by $t_0 = \min\{t' \in \mathbb{R} : [t', t] \subseteq tp(v)\}$.

If we fix a vertex $v$ and a departure time $t \in \mathbb{R}$, the arrival interval $I_v^\sigma(t)$ contains the points in time at which we are allowed to arrive at $v$. In the first case $t \notin tp(v)$ we are not allowed to depart at time $t$. Consequently, there is no time at which we can arrive in order to depart at the time $t$. If we are not allowed to wait at $v$, we have to arrive exactly at the time at which we want to depart. Finally, if we are allowed to wait, we must arrive not later than $t$ but in the time window of $tp(v)$ that contains $t$. This is expressed by the interval $[t_0, t]$.

**Definition 7.4** (Valid Path)**.** Let $(G, v_s, v_t, d, tp, W)$ be an FPTPW network based on a graph $G = (V, A)$ and let $u, v \in V$.

A *departure time specification* for a path $P = (v_0, \dots, v_k)$ is a function $\tau\colon V(P) \to \mathbb{R}$. The corresponding *arrival time specification* $\sigma\colon V(P) \to \mathbb{R}$ is defined by $\sigma(v_0) = \tau(v_0)$ and $\sigma(v_i) = \tau(v_{i-1}) + d(v_{i-1}v_i)$ for $i \in \{1, \dots, k\}$.

A tuple $(P, \tau)$ consisting of a $u$-$v$-path $P$ and a departure time specification $\tau$ for $P$ is a *valid $u$-$v$-path*, if the corresponding arrival times $\sigma$ satisfy $\sigma(w) \in I^{\sigma}_{w}(\tau(w))$ for all $w \in V(P)$. In this case we also call $\tau$ *valid for $P$*.

We simply call $\tau$ the *departure times* and $\sigma$ the *arrival times*. Note that arrival times always depend on departure times. However, we refrain from including the departure times in the notation for arrival times to keep the notation simple. Instead, when dealing with different departure times $\tau$ and $\tau'$, we analogously write $\sigma$ and $\sigma'$ for the corresponding arrival times. Furthermore, since we usually consider only valid paths, we sometimes omit the adjective *valid* and only refer to these as *paths*. It should always be clear from the context whether we need the corresponding departure times or not.

**Definition 7.5** (Fastest Path)**.** Let $(G, v_s, v_t, d, tp, W)$ be an FPTPW network based on a graph $G = (V, A)$. The *duration* $d(P, \tau)$ of a valid $u$-$v$-path $(P, \tau)$ is the difference $\tau(v) - \tau(u)$ between the departure times at its ends. A *fastest $u$-$v$-path* is a valid $u$-$v$-path $(P, \tau)$ with minimum duration $d(P, \tau)$.

We now have all ingredients needed to define the fastest path with time profiles and waiting problem, the central problem in this part of the thesis.

**Problem 7.6** (FPTPW)**.** An instance of the *Fastest Path with Time Profiles and Waiting* problem consists of an FPTPW network $(G, v_s, v_t, d, tp, W)$. In the optimization variant MinFPTPW the goal is to find a fastest $v_s$-$v_t$-path. The decision variant FPTPW requires another parameter $D \in \mathbb{R}$ and the question is whether there exists a valid $v_s$-$v_t$-path $(P, \tau)$ whose duration does not exceed $D$.

We close this first section about the problem FPTPW with a few assumptions that we use repeatedly throughout this thesis.

Vertices that are not contained in any path from the source to the target are irrelevant for the problem. We can identify and remove these vertices in linear time by performing two depth- or breadth-first-searches: one from the source $v_s$ in the original graph $G$ and one from the target $v_t$ in the inverse graph $G^{-1}$. Throughout this part, we assume that such vertices do not exist.

**Assumption 7.7.** Every vertex $v \in V$ is contained in some $v_s$-$v_t$-path.

Moreover, we often assume that the graph does not contain parallel arcs. This simplifies notation as we can associate outgoing arcs with the corresponding vertices. We can remove parallels by subdividing involved arcs. Replacing an arc $uw$ for that purpose means introducing a new vertex $v$ with the same properties (time profile and waiting characteristic) as $u$ and inserting the arcs $uv$ with $d(uv) = 0$ and $vw$ that gets assigned the duration of the original arc.

**Assumption 7.8.** The graph $G$ does not contain parallels.

Shifting the time profiles of all vertices by the same constant does not have much influence on the problem instance: we obtain a one-to-one correspondence between the valid paths of both instances by uniformly shifting their departure times. Compare also the *shifting* operation that we introduce later in Notation 7.17 on Page 84. In particular, the durations of corresponding paths coincide. Thus, we may assume that all time profiles contain only nonnegative time points, that is, $tp(v) \subseteq \mathbb{R}_{\geq 0}$ for all vertices $v \in V$.

**Assumption 7.9.** All time profiles contain only nonnegative time points.

## 7.2 Literature

Shortest path problems including a temporal component have already been extensively studied. This section gives an overview of the work that is particularly similar to our problem, at least in certain aspects. The problems are often inspired by applications from different thematic areas. We also refer to Section 3.3 for the literature that is more related to the practical problem on which FPTPW is based.

**Railway Routing** Halpern and Priess describe in [HP74] an almost identical problem, the setting is only slightly more general. They motivate this by scheduling trains on a railway network or convoys on a network of narrow streets. Every arc has a duration and a collection of time intervals representing permissible departure times. In addition, every vertex is associated with a finite set of time intervals during which parking is permissible. For two vertices $r$ and $s$, their goal is to find a valid $r$-$s$-path with minimum arrival time at the target $s$. This differs from FPTPW where we search for a path with fastest duration and therefore have to respect the departure time at the source. Consequently, Halpern and Priess can restrict to calculating the time points at which a valid path arrives or departs at a vertex. For this task they provide an algorithm, which is inspired by Dijkstra's shortest path algorithm [see Dij59]. By contrast, we cannot simply restrict these possible arrival and departure times but we have to further associate each such time point with the corresponding duration to get there (for more information see Chapter 9).

The problem from [HP74] is addressed again by Sancho in [San92] and [San94]. He restricts the number of time intervals associated with vertices or arcs to one, and gives dynamic programming formulations to solve the problem.

The situation dealt with in [HÁN20] is also quite similar to our practical problem. In fact, Haehn, Ábrahám, and Nießen consider basically the same task on a different data basis. They extend and adapt the algorithm of Halpern and Priess from [HP74] to their needs. In particular, they also have a fixed starting time $t_{start}$ to which the duration of the fastest path is related. This is the crucial point where our problem differs and why we need different algorithms. For further information we also refer to the literature overview for the practical problem in Section 3.3 from Page 13 on.

**Vehicle Scheduling** There are several publications on related problems in the field of vehicle scheduling. A common setting contains vehicles with one or multiple depots and rides, each associated with a time window in which it must be started. The task is to assign the rides to the vehicles and a subproblem is the routing of a single vehicle. This is often formalized as a shortest path problem with time windows and waiting.

Such a vehicle scheduling problem with multiple depots is described by Desaulniers, Lavigne, and Soumis in [DLS98]. In addition to the duration, each arc has a cost. Vehicles are allowed to wait everywhere and the time windows only restrict the departure times. More precisely, it is possible to arrive at a vertex before its time window opens and then wait a correspondingly long time. In [DV00], Desaulniers and Villeneuve extract the subproblem of scheduling a single vehicle and extend the model by linear waiting costs. This Shortest Path with Waiting Costs (SPWC) problem is similar to our problem when we choose costs equal to durations. However, waiting is still allowed everywhere, and one can still arrive before the time windows open.

A similar vehicle scheduling problem is described by Desrosiers, Soumis, and Desrochers in [DSD84]. As with the above mentioned SPWC, the Shortest Path with Time Windows (SPTW) subproblem described therein is closely related to our problem. Desrochers and Soumis present another algorithm which solves this problem in [DS88]. However, as for SPWC, also SPTW allows waiting everywhere and arriving before time windows open.

**Postal Services** In the route planning for postal services, time windows are common constraints, especially when delivering parcels to industry customers. These problems are often modeled as traveling salesperson problems, see, for example [BDR21].

**Propagating Functions** The main idea of our algorithms solving FPTPW is to propagate functions, which is also done by Xing and Kao in [XK02] for a different problem. They want to find a shortest path in the plane with rectangular obstacles. Based on the obstacles they divide the plane into segments and compute shortest path functions for the boundaries of these segments. That is, their functions depend on space and refer to shortest distances where our functions depend on time and refer to fastest durations. However, their functions are also piecewise linear and the algorithmic idea is similar.

## 7.3 Handling Paths

The aim of this section is to provide tools to handle (valid) paths. First, we give an alternative characterization for the validity of departure times. Thereupon we consider a few variants of modifying paths: restricting and concatenating paths, replacing subpaths, and shifting the departure times slightly.

**Lemma 7.10.** Let $(G, v_s, v_t, d, tp, W)$ be an FPTPW network and let $P$ be a path in $G$. A departure time specification $\tau\colon V(P) \to \mathbb{R}$ is valid if and only if there exist time windows $I_v \subseteq tp(v)$ for the vertices on the path $P$ such that the following conditions are satisfied:

(DT 1) $\tau(v) \in I_v$ for all $v \in V(P)$,

(DT 2) $\sigma(v) \in I_v$ for all $v \in V(P)$,

(DT 3) $\tau(v) \geq \sigma(v)$ for all $v \in V(P)$, and

(DT 4) $\tau(v) = \sigma(v)$ for all $v \in V(P) \setminus W$.

*Proof.* First, let $\tau$ be a valid departure time specification and let $v \in V$. Since $\tau$ is valid, the arrival time $\sigma(v) \in I_v^\sigma(\tau(v))$ is contained in the arrival interval. Thus, the arrival interval $I_v^\sigma(\tau(v))$ is non-empty, which according to Definition 7.3 can only be the case if $\tau(v) \in tp(v)$. Let $I_v$ be the time window of $tp(v)$ containing $\tau(v)$. Again by the definition of the arrival interval, it is $I_v^\sigma(\tau(v)) \subseteq I_v$ and every $t \in I_v^\sigma(\tau(v))$ satisfies $t \leq \tau(v)$. This implies (DT 1) to (DT 3). In the case $v \notin W$, the arrival interval $I_v^\sigma(\tau(v)) = \{\tau(v)\}$ only consists of a single point and $\sigma(v) \in I_v^\sigma(\tau(v))$ also implies (DT 4).

Now let $\tau\colon V(P) \to \mathbb{R}$ and let $I_v \subseteq tp(v)$ be some time windows for the vertices $v \in V(P)$ that satisfy (DT 1) to (DT 4). Let $v \in V(P)$. By (DT 1) we have that the arrival interval $I_v^\sigma(\tau(v)) \neq \varnothing$ is non-empty. In the case $v \notin W$, (DT 4) implies $\sigma(v) = \tau(v) \in I_v^\sigma(\tau(v))$. Otherwise, it follows from (DT 1) to (DT 3) and the fact that $I_v$ is a time window that $[\sigma(v), \tau(v)] \subseteq tp(v)$. Hence, it is $\sigma(v) \in I_v^\sigma(\tau(v))$ by Definition 7.3. □

As already mentioned above, we want to provide operations to modify valid paths. Common paths can be restricted or concatenated and subpaths can be replaced. We translate these operations to valid paths, making sure the temporal components fit together.

## Restriction

We can restrict every valid path to a subpath by restricting the spatial path as usual and by keeping only the departure times of vertices that remain on the restricted path.

**Definition 7.11** (Restriction)**.** Let $(P, \tau)$ be a valid path and let $\{u, v\} \subseteq V(P)$ be two vertices on this path. *Restricting* $(P, \tau)$ to its $u$-$v$-subpath results in the path $(P, \tau)|_{uv} = (P|_{uv}, \tau|_{V(P|_{uv})})$.

**Lemma 7.12.** Restricting a valid path yields another valid path.

*Proof.* Let $(P', \tau') = (P, \tau)|_{uv}$ be a restriction of a valid path to its $u$-$v$-subpath. By Definition 7.4 the path $(P', \tau')$ is valid if its arrival times $\sigma'$ satisfy $\sigma'(w) \in I_w^\sigma(\tau'(w))$ for all $w \in V(P')$. The arrival times $\sigma$ and $\sigma'$ of the path and its restriction can differ at most in the vertex $u$ as $\sigma(u) \leq \tau(u) = \tau'(u) = \sigma'(u)$. The claim now follows since $(P, \tau)$ is valid and the arrival intervals are independent of the path. □

## Concatenation

In order to concatenate two valid paths, their spatial and temporal information must match. On the one hand, the end-vertex of the first path must be the start-vertex of the second path. On the other hand, the first path has to arrive at its end-vertex within the arrival interval corresponding to the departure time of the second path.

**Definition 7.13** (Concatenation)**.** Let $(P, \tau)$ be a valid $u$-$v$-path and let $(P', \tau')$ be a valid $v$-$w$-path with $\sigma(v) \in I_v^\sigma(\tau'(v))$. *Concatenating* $(P, \tau)$ and $(P', \tau')$ results in the path $(P, \tau) \circ (P', \tau') = (P \circ P', \tau'')$ with $\tau'' \colon V(P \circ P') \to \mathbb{R}$ defined by

$$\tau''(x) = \begin{cases} \tau'(x) & \text{if } x \in V(P') \text{ and} \\ \tau(x) & \text{otherwise.} \end{cases}$$

**Lemma 7.14.** Concatenating two valid paths yields another valid path.

*Proof.* Let $(P'', \tau'') = (P, \tau) \circ (P', \tau')$ be the concatenation of a valid $u$-$v$-path and a valid $v$-$w$-path. We have to show that $\sigma''(x) \in I_x^\sigma(\tau''(x))$ for all $x \in V(P'')$. If $x \neq v$, this follows since $(P, \tau)$ and $(P', \tau')$ are valid. In the case $x = v$ this follows as $\sigma''(v) = \sigma(v)$, $\tau''(v) = \tau'(v)$, and because the concatenation requires $\sigma(v) \in I_v^\sigma(\tau'(v))$. □

## Replacement

To replace the subpath of a valid path by another valid path, the spatial and temporal information on both ends of the subpath have to fit. As we can replace a part at the start, at the end, or right in the middle, we need a case distinction. Since a replacement is nothing but the concatenation of (restricted) paths, we only have to ensure that the corresponding operations can be performed.

**Definition 7.15** (Replacement)**.** Let $(P, \tau)$ be a valid $u$-$v$-path and let $(P', \tau')$ be a valid $u'$-$v'$-path with $\{u', v'\} \subseteq V(P)$ such that $u'$ precedes $v'$ on $P$. If $u' = u$ or $\sigma(u') \in I^{\sigma}_{u'}(\tau'(u'))$, and $v' = v$ or $\sigma'(v') \in I^{\sigma}_{v'}(\tau(v'))$, *replacing* the $u'$-$v'$-subpath of $(P, \tau)$ by the path $(P', \tau')$ results in the path

$$\begin{cases} (P,\tau)|_{uu'} \circ (P',\tau') \circ (P,\tau)|_{v'v} & \text{if } u \neq u' \text{ and } v \neq v', \\ (P,\tau)|_{uu'} \circ (P',\tau') & \text{if } u \neq u' \text{ and } v = v', \\ (P',\tau') \circ (P,\tau)|_{v'v} & \text{if } u = u' \text{ and } v \neq v', \text{ and} \\ (P',\tau') & \text{if } u = u' \text{ and } v = v'. \end{cases}$$

**Lemma 7.16.** Replacing a subpath of a valid path by a valid path as specified in Definition 7.15 yields another valid path.

*Proof.* The requirements for the replacement are chosen such that the restrictions and concatenations of the different cases are feasible. Thus, the claim follows by Lemmas 7.12 and 7.14. □

## Shifting

We can shift a path by uniformly adding or subtracting some constant from all departures times. This results in a path with the same duration that departs and arrives this constant later or earlier, respectively. Although we can shift a path by arbitrarily large constants, we are mostly interested in small shifts.

**Notation 7.17.** Let $(P, \tau)$ be some $u$-$v$-path and let $\varepsilon > 0$. *Shifting* the path by $\varepsilon$ *forward in time* means that we switch to the path $(P, \tau + \varepsilon)$. Analogously, *shifting* it by $\varepsilon$ *backward in time* results in the path $(P, \tau - \varepsilon)$.

**Lemma 7.18.** Let $(P, \tau)$ be a $u$-$v$-path and let $\delta > 0$. The path $(P, \tau + \varepsilon)$ is valid for all $0 \leq \varepsilon \leq \delta$ if and only if $[\tau(w), \tau(w) + \delta] \subseteq tp(w)$ for all $w \in V(P)$.

*Proof.* By (DT 1) from Lemma 7.10 we have $\tau(w) + \varepsilon \in tp(w)$ if $(P, \tau + \varepsilon)$ is valid. Thus, the validity of $(P, \tau + \varepsilon)$ for $0 \leq \varepsilon \leq \delta$ implies $[\tau(w), \tau(w) + \delta] \subseteq tp(w)$ for all $w \in V(P)$.

Assuming now that $[\tau(w), \tau(w) + \delta] \subseteq tp(w)$ for all $w \in V(P)$, we show that $(P, \tau + \varepsilon)$ satisfies (DT 1) to (DT 4) from Lemma 7.10 for every $0 \leq \varepsilon \leq \delta$. As the arrival times of $(P, \tau + \varepsilon)$ are exactly the arrival times $\sigma$ of $(P, \tau)$ shifted by $\varepsilon$, the difference of arrival and departure at every vertex remains the same. Hence, (DT 3) and (DT 4) are satisfied. Now, let $w \in V$ and $0 \leq \varepsilon \leq \delta$. (DT 1) and (DT 2) hold because $[\sigma(w), \tau(w) + \varepsilon] \subseteq tp(w)$. Therefore, the same time window $I_w \subseteq tp(w)$ that contains the original arrival and departure also contains the shifted arrival and departure. □

**Lemma 7.19.** Let $(P, \tau)$ be a $u$-$v$-path with arrival times $\sigma$ and let $\delta > 0$. The path $(P, \tau - \varepsilon)$ is valid for all $0 \leq \varepsilon \leq \delta$ if and only if $[\sigma(w) - \delta, \sigma(w)] \subseteq tp(w)$ for all $w \in V(P)$.

*Proof.* Similar to the proof of Lemma 7.18. □

## 7.4 MIP Formulation

The problems FPTPW and MINFPTPW can both be formulated by mixed integer linear programs. Thus, let $(G = (V, A), v_s, v_t, d, tp, W)$ be an FPTPW network. For the programming formulation we introduce the following variables:

▷ A binary variable $y(uv)$ for every arc $uv \in A$ that states whether $uv$ is on the fastest path.

▷ A binary variable $x(v, I)$ for every vertex $v \in V$ and each of its time windows $I \subseteq tp(v)$. It describes whether the fastest path uses this time window.

▷ A continuous variable $\tau(v)$ for every vertex $v \in V$ that denotes the departure time at this vertex if it is on the fastest path.

For ease of notation we introduce further variables that can be eliminated again:

▷ A binary variable $x(v) = \sum_{I \in tp(v)} x(v, I)$ for every vertex $v \in V$ stating whether it is used by the fastest path.

▷ Variables $t_0(v) = \sum_{I \in tp(v)} \min(I) \cdot x(v, I)$ and $t_1(v) = \sum_{I \in tp(v)} \max(I) \cdot x(v, I)$ attaining lower and upper bounds on $\tau(v)$ for every vertex $v \in V$.

By $M$ we denote a time constant that is large enough to guarantee that Constraints (7.1k) to (7.1m) are fulfilled for every arc $uv \in A$ with $y(uv) = 0$, that is, for every arc not on the fastest path. If we assume that all time profiles contain only nonnegative time points (Assumption 7.9 applies), it suffices to choose this constant $M$ larger than $\max_{v \in V} \max(tp(v))$. Otherwise, we also have to respect the difference between the largest and the smallest time point across all time profiles.

With these variables we can formulate MinFPTPW as mixed integer linear program:

$$
\begin{alignedat}{3}
\min \quad & \tau(v_t) - \tau(v_s) & & \qquad && (7.1\text{a})\\
\text{s.t.} \quad & \sum_{u \in N^{\text{out}}(v_s)} y(v_s u) = 1 & & && (7.1\text{b})\\
& \sum_{u \in N^{\text{in}}(v)} y(uv) = \sum_{u \in N^{\text{out}}(v)} y(vu) & \quad & \forall\, v \in V \setminus \{v_s, v_t\} && (7.1\text{c})\\
& x(v_s) = 1 & & && (7.1\text{d})\\
& x(v) = \sum_{u \in N^{\text{in}}(v)} y(uv) & & \forall\, v \in V \setminus \{v_s\} && (7.1\text{e})\\
& x(v) = \sum_{I \in tp(v)} x(v,I) & & \forall\, v \in V && (7.1\text{f})\\
& t_0(v) = \sum_{I \in tp(v)} \min(I) \cdot x(v,I) & & \forall\, v \in V && (7.1\text{g})\\
& t_1(v) = \sum_{I \in tp(v)} \max(I) \cdot x(v,I) & & \forall\, v \in V && (7.1\text{h})\\
& \tau(v) \geq t_0(v) & & \forall\, v \in V && (7.1\text{i})\\
& \tau(v) \leq t_1(v) & & \forall\, v \in V && (7.1\text{j})\\
& t_0(v) \leq \tau(u) + d(uv) \cdot y(uv) + M \cdot (1 - y(uv)) & & \forall\, uv \in A && (7.1\text{k})\\
& \tau(v) \geq \tau(u) + d(uv) \cdot y(uv) - M \cdot (1 - y(uv)) & & \forall\, uv \in A && (7.1\text{l})\\
& \tau(v) \leq \tau(u) + d(uv) \cdot y(uv) + M \cdot (1 - y(uv)) & & \forall\, uv \in A : u \notin W && (7.1\text{m})\\
& x(v) \in \{0,1\} & & \forall\, v \in V && (7.1\text{n})\\
& x(v,I) \in \{0,1\} & & \forall\, v \in V\ \forall\, I \in tp(v) && (7.1\text{o})\\
& y(uv) \in \{0,1\} & & \forall\, uv \in A && (7.1\text{p})
\end{alignedat}
$$

The Constraints (7.1b) and (7.1c) ensure that the arcs selected by the $y$ variables correspond to a $v_s$-$v_t$-path in $G$. With Constraints (7.1d) and (7.1e) we enforce the $x$ variables to select exactly the vertices on the path defined by $y$. Constraint (7.1f) guarantees that we select exactly one time window for every vertex on the path. From Constraints (7.1g) and (7.1h) we get the time bounds of the specified time windows. The Constraints (7.1i) and (7.1j) restrict the departure times $\tau(v)$ to the selected time intervals and with Constraint (7.1k) we ensure that the arrival times are not earlier than the selected time windows. The Constraints (7.1l) and (7.1m) enforce that we depart later than the arrival time or at the arrival time if waiting is not allowed.

Note that we can remove Constraints (7.1f) to (7.1h) if we substitute the variables $x(v)$, $t_0(v)$, and $t_1(v)$ appropriately.

## 7.5 Hardness

**Theorem 7.20.** FPTPW is **NP**-complete.

*Proof.* Given a path $P$ together with departure times $\tau$, we can check whether $(P, \tau)$ is a valid path in polynomial time. We only have to ensure that $P$ is in fact a $v_s$-$v_t$-path, that the departure times $\tau$ satisfy (DT 1) to (DT 4) from Lemma 7.10, and that the duration $\tau(v_t) - \tau(v_s)$ does not exceed the maximum duration $D$. Since each of these tasks only takes polynomial time, FPTPW is contained in **NP**.

To prove the hardness, we present a reduction from the **NP**-complete SUBSETSUM problem [GJ79, Problem SP13]. For $B \in \mathbb{N}_{>0}$ and $b_1, \ldots, b_k \in \mathbb{N}_{>0}$ the question is whether a set $J \subseteq \{1, \ldots, k\}$ with $B = \sum_{i \in J} b_i$ exists. We construct an FPTPW network based on the graph depicted in Figure 7.1. It is a path with vertices $v_0, \ldots, v_k$ whose arcs are doubled: for $i = 1, \ldots, k$ the graph contains two arcs from $v_{i-1}$ to $v_i$. One with a duration of $b_i$ that we denote by $(v_{i-1}v_i)^+$ and the other with a duration of 0 that we denote by $(v_{i-1}v_i)^0$. The time profiles consist of single time windows each. The source $v_s = v_0$ gets $tp(v_0) = [0, 0]$, the target $v_t = v_k$ gets $tp(v_k) = [B, B]$, and every remaining vertex $v_i$ (with $1 \le i < k$) gets $tp(v_i) = [0, B]$. Waiting is not allowed at any vertex, that is, $W = \varnothing$.

Every $v_0$-$v_k$-path $P$ uses for $i = 1, \ldots, k$ exactly one of the two parallel arcs $(v_{i-1}v_i)^+$ and $(v_{i-1}v_i)^0$. This allows us to uniquely identify such a path $P$ with a subset $J \subseteq \{1, \ldots, k\}$: $i \in J$ if and only if $P$ uses $(v_{i-1}v_i)^+$. All valid departure times satisfy $\tau(v_k) - \tau(v_0) = B$ as $\tau(v_k) \in tp(v_k) = \{B\}$ and $\tau(v_0) \in tp(v_0) = \{0\}$. Thus, the valid paths correspond exactly to the subsets of $\{b_1, \ldots, b_k\}$ summing up to $B$. □

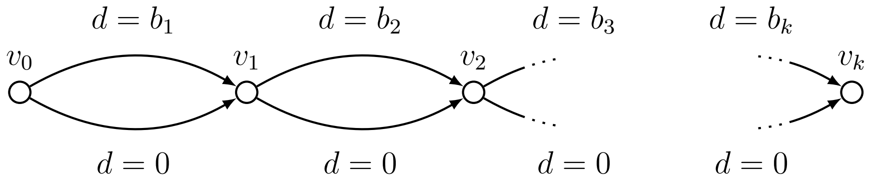


Figure 7.1: Graph with durations of the FPTPW network for the proof of Theorem 7.20.

Note that the graph constructed in the proof of Theorem 7.20 and visualized in Figure 7.1 contains parallels. However, we can eliminate these by subdividing, for example, every arc $(v_{i-1}v_i)^0$ as described directly before Assumption 7.8 on Page 80.

The FPTPW network defined for the reduction in the proof of Theorem 7.20 is based on a quite simple graph and does not make use of waiting. Thus, we can tighten the **NP**-completeness claim which we state in Corollaries 7.21 and 7.22 for future reference.

**Corollary 7.21.** FPTPW is still **NP**-complete under the restriction that waiting is forbidden everywhere.

**Corollary 7.22.** FPTPW is still **NP**-complete if the underlying graph is a path whose arcs are doubled. In particular, it is **NP**-complete for series-parallel graphs.

Corollary 7.21 suggests that the prohibition of waiting makes the problem hard whereas Corollary 7.22 refers to the structure of the graph. We further examine the complexity of the problem later in this part. For example, we show in Corollary 11.6 that the basic algorithm to solve MinSFP (Algorithm 10.1 from Section 10.2) runs in polynomial time if waiting is allowed everywhere. We summarize the complexity-related results in Chapter 12.

## Conclusion

With the fastest path with time profiles and waiting problem we formulated a simplified, one-dimensional version of the practical problem from Section 3.5, which retains the basic structure but omits technical refinements. Its feasible solutions are valid paths and we defined different ways to modify and combine them: restriction, concatenation, replacement, and shifting. Except for the last one, these are essentially standard operations on paths that have been generalized for our time-dependent setting. Furthermore, we presented a MIP formulation of the problem and proved that FPTPW is **NP**-complete.

In what follows, we inspect the problem in more detail and start with the time profiles. We show how to reduce them to the necessary parts and gain first knowledge about the hard parts of this problem.

Chapter 8

# Synchronizing Time Profiles

*Not every time point in the time profiles needs to be contained in a valid path. If this is the case, we can shrink the time profiles to the necessary parts. In this section, we present an algorithm to perform this shrinking. Its execution automatically synchronizes the time profiles but their number might grow exponentially. We analyze these characteristics and determine sufficient conditions for instances whose increase in size remains polynomially bounded.*

## Assumptions

Throughout this chapter, an FPTPW network $N = (G, v_s, v_t, d, tp, W)$ based on a graph $G = (V, A)$ is always given. The following assumptions apply.

**Only Relevant Vertices** → see Assumption 7.7 on Page 79
Every vertex is contained in a $v_s$-$v_t$-path.

**No Parallels** → see Assumption 7.8 on Page 80
The graph does not contain parallel arcs.

## 8.1 Idea and Algorithm

If a time point $t$ in the time profile of a vertex $v$ cannot be used by any valid path, it is essentially irrelevant for the problem. "Cannot be used" in this case means that there is no valid path $(P, \tau)$ satisfying $\sigma(v) \leq t \leq \tau(v)$. Thus, the set of feasible paths remains the same if we remove all of these time points. Because we assume in the following chapters that such time points does not exist, we formulate this in Assumption 8.1. The goal of this chapter is to develop an algorithm that shrinks the time profiles of a given instance such that every valid path remains valid but this assumption is fulfilled.

**Assumption 8.1.** For any vertex $v \in V$ and any time point $t \in tp(v)$ there exists a valid $v_s$-$v_t$-path $(P, \tau)$ that contains $v$ and satisfies $\sigma(v) \leq t \leq \tau(v)$.

An obvious question is whether the time profiles retain the property that they can be written as a union of finitely many closed intervals, which is required by Definition 7.1. If this is the case, the question arises whether the number of time windows increases significantly. We can answer the first one positively in Corollary 8.5. However, the number of time windows can grow exponentially by the synchronization. We examine this in detail in Section 8.3.

Before we go into the algorithm, we gather a few reasons why this approach of shrinking respectively synchronizing the time profiles makes sense in the first place. On the one hand, we can use this method to figure out whether a valid path exists at all. If we are only interested in this question, there is no need to consider fastest durations. Although this synchronization of the time profiles goes a little beyond the question of whether a valid path exists, there is probably no much simpler way to prove that there is no valid path. On the other hand, this shrinking reduces the search space for fastest path algorithms solving FPTPW. As mentioned above, this can be helpful since an FPTPW algorithm should need to store much more information. For every time point that is possibly on a valid path, such an algorithm has to keep information about the duration to get there. In particular, every FPTPW algorithm has to do this synchronization, at least implicitly. Outsourcing this step to a simpler, preceding algorithm keeps us from expensively storing unnecessary information about fastest durations. A disadvantage in practical applications is that we might waste a lot of resources on time intervals that are not on fastest paths at all. In terms of worst case analysis, however, this does not make a difference. This can already be seen in the **NP**-completeness proof for FPTPW, see Theorem 7.20. Valid paths for the constructed instances therein all have the same duration. Thus, FPTPW is already hard if we are only interested in whether a valid path exists.

We now describe an algorithm to ensure Assumption 8.1, which iterates over the vertices of the graph twice. During the first pass (the *forward iteration*) it removes those parts of the time profiles that cannot be reached from the source $v_s$. This is done by propagating the intervals which can be reached from the source throughout the graph to the target. Similarly, in the second pass (the *backward iteration*) it removes those parts of the time profiles from which we can no longer reach the target $v_t$. It is basically the same as the forward iteration in "the inverse graph with reversed time".

When considering a vertex $v$, we have to distinguish whether or not waiting is allowed there. If waiting is forbidden, we can simply shift the time profiles of the predecessors of $v$ (respectively the successors in the backward iteration) by the respective durations and intersect $tp(v)$ with the union of these. Otherwise, if waiting is allowed at $v$, for every time window $I \subseteq tp(v)$ we compute the earliest possible arrival time and cut off everything before. In the backward iteration this earliest arrival time transfers to the latest departure time and we cut off the remaining part of the respective time window.

The whole procedure is formalized in Algorithm 8.1. Note that $v_s = v_0$ and $v_t = v_{n-1}$ since we assume that Assumption 7.7 applies.

**Algorithm 8.1:** Synchronizing time profiles

**Input:** FPTPW network $N = (G, v_s, v_t, d, tp, W)$ with $G = (V, A)$, $n = |V|$
**Output:** Synchronized time profiles $\overleftrightarrow{tp}\colon V \to \mathcal{T}$

```
Label the vertices according to a topological ordering of $G$ with $v_0, v_1, \dots, v_{n-1}$.
// Forward iteration
$\overrightarrow{tp}(v_0) \coloneqq tp(v_0)$
for $i = 1, \dots, n-1$ do
    $tp_{\text{preds}} \coloneqq \bigcup_{u \in N^{\text{in}}(v_i)} (\overrightarrow{tp}(u) + d(uv_i))$
    if $v_i \notin W$ then                                        // waiting forbidden
        $\overrightarrow{tp}(v_i) \coloneqq tp(v_i) \cap tp_{\text{preds}}$
    else                                                          // waiting allowed
        $\overrightarrow{tp}(v_i) \coloneqq \varnothing$
        for every time window $I \subseteq tp(v_i)$ do
            $\overrightarrow{tp}(v_i) \coloneqq \overrightarrow{tp}(v_i) \cup (I \cap ((I \cap tp_{\text{preds}}) + [0, \infty)))$.
        end
    end
end
// Backward iteration
$\overleftrightarrow{tp}(v_{n-1}) \coloneqq \overrightarrow{tp}(v_{n-1})$
for $i = n-2, \dots, 0$ do
    $tp_{\text{succs}} \coloneqq \bigcup_{u \in N^{\text{out}}(v_i)} (\overleftrightarrow{tp}(u) - d(v_i u))$
    if $v_i \notin W$ then                                        // waiting forbidden
        $\overleftrightarrow{tp}(v_i) \coloneqq \overrightarrow{tp}(v_i) \cap tp_{\text{succs}}$
    else                                                          // waiting allowed
        $\overleftrightarrow{tp}(v_i) \coloneqq \varnothing$
        for every time window $I \subseteq \overrightarrow{tp}(v_i)$ do
            $\overleftrightarrow{tp}(v_i) \coloneqq \overleftrightarrow{tp}(v_i) \cup (I \cap ((I \cap tp_{\text{succs}}) - [0, \infty)))$.
        end
    end
end
return $\overleftrightarrow{tp}$
```

## 8.2 Correctness

The correctness of Algorithm 8.1 is proved in Theorem 8.4, for which the following two lemmas serve as preparation. Essentially, Lemmas 8.2 and 8.3 provide the basic results that we need to prove that the resulting time profiles satisfy Assumption 8.1. What remains for the proof of Theorem 8.4 is to show that we do not lose any valid path.

**Lemma 8.2.** Let $N$ be an FPTPW network and let $\overleftrightarrow{tp}$ be the synchronized time profiles generated by Algorithm 8.1 on input $N$. Furthermore, let $v \in V \setminus \{v_s\}$, $I \subseteq \overleftrightarrow{tp}(v)$, and $t \in I$. Then, the following holds:

- ▷ If $v \notin W$, there exist $u \in N^{\text{in}}(v)$ and $t' \in \overleftrightarrow{tp}(u)$ such that $t = t' + d(uv)$.
- ▷ If $v \in W$, there exist $u \in N^{\text{in}}(v)$, $t' \in \overleftrightarrow{tp}(u)$, and $t'' \in I \cap (-\infty, t]$ such that $t'' = t' + d(uv)$.

*Proof.* First, we prove that the claim holds for $\overrightarrow{tp}$ instead of $\overleftrightarrow{tp}$ after the execution of the `for`-loop in Lines 3 to 13 (the forward iteration). Second, we show that all actions within the `for`-loop in Lines 15 to 25 (the backward iteration) preserve the stated properties, that is, the claimed properties afterwards also hold for $\overleftrightarrow{tp}$.

*Step 1.* Let $v \in V \setminus \{v_s\}$, $I \subseteq \overrightarrow{tp}(v)$ be a time window in $\overrightarrow{tp}(v)$, and $t \in I$. By $v \neq v_s$ and Assumption 7.7 we have $N^{\text{in}}(v) \neq \varnothing$. If $v \notin W$, we have $t \in tp_{\text{preds}}$ and by its definition in Line 4 there exists $u \in N^{\text{in}}(v)$ and $t' \in \overrightarrow{tp}(u)$ with $t = t' + d(uv)$. If $v \in W$, we have $t \in I' \cap ((I' \cap tp_{\text{preds}}) + [0, \infty))$ for some time window $I' \subseteq tp(v)$. Hence, $t = t'' + c$ for $t'' \in I' \cap tp_{\text{preds}}$ and $c \geq 0$. Due to the maximality of the time window $I'$, $[t'', t] \subseteq I'$, and $[t'', t] = [t'', t''] + [0, c]$ we have $t'' \in I \cap (-\infty, t]$. Since $t'' \in tp_{\text{preds}}$, there exist $u \in N^{\text{in}}(v)$ and $t' \in \overrightarrow{tp}(u)$ with $t'' = t' + d(uv)$.

Note that this argumentation suffices since the time profile $\overrightarrow{tp}(u)$ of a vertex $u$ is only modified in a single iteration of the `for`-loop and since $\overrightarrow{tp}(u)$ for $u \in N^{\text{in}}(v)$ is modified earlier than $\overrightarrow{tp}(v)$.

*Step 2.* Let $v \in V \setminus \{v_s\}$, $I \subseteq \overleftrightarrow{tp}(v)$ be a time window in $\overleftrightarrow{tp}(v)$, and $t \in I$. We have $\overleftrightarrow{tp}(v) \subseteq \overrightarrow{tp}(v)$ and thus also $t \in \overrightarrow{tp}(v)$. Furthermore, $I \subseteq I'$ for a time window $I'$ in $\overrightarrow{tp}(v)$. If $v \notin W$, let $t'' = t$ and if $v \in W$, let $t''$ as defined in step 1. In the latter case we have $[t'', t] \subseteq I'$. When considering $I'$ in the loop in Line 21 of Algorithm 8.1 we add not only $t$ to $\overleftrightarrow{tp}(v)$ but also all points of $I'$ which are less than $t$. In particular, we have $[t'', t] \subseteq I$. Hence, $t'' \in I \cap (-\infty, t]$ as required.

Now, let $u \in N^{\text{in}}(v)$ and $t' \in \overrightarrow{tp}(u)$ as defined in step 1 with the property $t'' = t' + d(uv)$. It remains to prove that $t' \in \overleftrightarrow{tp}(v)$. However, since $t'' \in \overleftrightarrow{tp}(v)$ and $v \in N^{\text{out}}(u)$ we have $t' = t'' - d(uv) \in tp_{\text{succs}}$ and Algorithm 8.1 enforces in both cases $tp_{\text{succs}} \subseteq \overleftrightarrow{tp}(v)$. □

**Lemma 8.3.** Let $N$ be an FPTPW network and $\overleftrightarrow{tp}$ the synchronized time profiles generated by Algorithm 8.1 on input $N$. Furthermore, let $v \in V \setminus \{v_t\}$, $I \subseteq \overleftrightarrow{tp}(v)$, and $t \in I$. Then, the following holds:

▷ If $v \notin W$, there exist $u \in N^{\text{out}}(v)$ and $t' \in \overleftrightarrow{tp}(u)$ such that $t' = t + d(vu)$.
▷ If $v \in W$, there exist $u \in N^{\text{out}}(v)$, $t' \in \overleftrightarrow{tp}(u)$, and $t'' \in I \cap [t, \infty)$ such that $t' = t'' + d(vu)$.

*Proof.* Analogous to the proof of Lemma 8.2. □

**Theorem 8.4.** Algorithm 8.1 computes time profiles that satisfy Assumption 8.1 and for which any valid departure time specification for a $v_s$-$v_t$-path remains valid.

*Proof.* The first part of the claim follows by iterated applying and combining Lemmas 8.2 and 8.3. To prove the second part, let $(P, \tau)$ be a valid path before Algorithm 8.1 is executed. We show that $\tau$ is still a valid departure time specification afterwards. For this we show that $\tau$ is valid with respect to the time profiles $\overrightarrow{tp}$, which are calculated as intermediate results in the algorithm. Analogous to this reasoning, we can then show that $\tau$ is also a valid departure time specification for the time profiles $\overleftrightarrow{tp}$.

Let $P = (v_0, v_1, \dots, v_k)$. We show by induction on the vertices of $P$ that there are time windows $\hat{I}_v \subseteq \overrightarrow{tp}(v)$ for $v \in V(P)$ fulfilling the conditions from Lemma 7.10. Since the last two properties (DT 3) and (DT 4) only depend on $\tau$ and not on the time profiles, we only have to show the first two properties (DT 1) and (DT 2). Let $I_v \in tp(v)$ for $v \in V(P)$ be the time profiles proving that $\tau$ is a valid departure time specification with respect to $tp$.

For the source $v_s = v_0$ we can choose $\hat{I}_{v_s} = I_{v_s}$ since $\overrightarrow{tp}(v_s) = tp(v_s)$. Now consider $v_i$ for $i > 0$ and assume that appropriate time windows $\hat{I}_{v_j}$ for $j < i$ are already chosen. We first consider $v_i \notin W$. In this case $\tau(v_i) = \tau(v_{i-1}) + d(v_{i-1}v_i)$ and we only have to show that $\tau(v_i) \in \overrightarrow{tp}(v_i)$ ($\hat{I}_{v_i}$ is then the (unique) time window containing $\tau(v_i)$). However, since $\tau(v_{i-1}) \in \overrightarrow{tp}(v_{i-1})$ we have that $\tau(v_i) \in tp_{\text{preds}}$ in Line 4 of Algorithm 8.1. Hence, $\tau(v_i) \in I_{v_i} \cap tp_{\text{preds}}$ will be added to $\overrightarrow{tp}(v_i)$. In the case $v_i \in W$ we get by the same argumentation that $\tau(v_{i-1}) + d(v_{i-1}v_i) \in I_{v_i} \cap tp_{\text{preds}}$. Since $\tau$ is valid for $tp$ we also have $[\tau(v_{i-1}) + d(v_{i-1}v_i), \tau(v_i)] \subseteq I_{v_i}$. Thus, the interval added to $\overrightarrow{tp}(v_i)$ in Line 10 contains at least the interval $[\tau(v_{i-1}) + d(v_{i-1}v_i), \tau(v_i)]$. □

**Corollary 8.5.** The synchronized time profiles that ensure Assumption 8.1 can be written as a union of finitely many closed intervals.

*Proof.* Algorithm 8.1 ensures Assumption 8.1, and during the execution of the algorithm all time profiles remain unions of finitely many closed intervals. □

## 8.3 Running Time

Before we analyze the running time of Algorithm 8.1 in detail in Theorem 8.8, we present an example that shows its output sensitivity. However, the running time of Algorithm 8.1 actually depends not only of its output $\overleftrightarrow{tp}$, but also on the internal time profiles $\overrightarrow{tp}$ as we explain in Remark 8.9.

**Example 8.6.** For certain instances Algorithm 8.1 divides the intervals exponentially often. The graph of an FPTPW network corresponding to such an instance together with the arc durations is shown in Figure 8.1. Note that we use parallel arcs for the sake of clarity. We can remove the parallels by subdividing, for example, the arcs with duration 0 as described directly before Assumption 7.8 on Page 80. We have $v_s = v_0$ and $v_t = v_{n-1}$. The time profile for vertex $v_i$ consists of the single time window $[0, 2^{i+1} - 1]$ and we set $W = \varnothing$. Algorithm 8.1 splits the time profile of a vertex $v_i$ into the $2^i$ disjoint time windows $[0, 1], [2, 3], [4, 5], \dots, [2^{i+1} - 2, 2^{i+1} - 1]$. $\triangleleft$

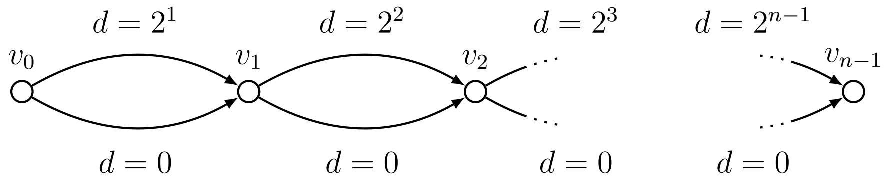


Figure 8.1: Graph with durations of the FPTPW network for Example 8.6.

A key point for the exponential increase of time windows in Example 8.6 is the prohibition of waiting ($W = \varnothing$). In fact, synchronizing the time intervals does not increase the number of time windows for vertices where waiting is allowed. We only move the lower bound to the earliest arrival time and the upper bound to the latest departure time. This is formalized and proved in the following lemma.

**Lemma 8.7.** Let $N$ be an FPTPW network, $\overleftrightarrow{tp}$ the synchronized time profiles generated by Algorithm 8.1 on input $N$, and let $v \in V$ be any vertex. If $v \in W$, then it holds $|\overleftrightarrow{tp}(v)| \leq |\overrightarrow{tp}(v)| \leq |tp(v)|$.

*Proof.* For a vertex $v$ with $v \in W$, Algorithm 8.1 restricts all time windows $I \subseteq tp(v)$ in Line 10. However, only the lower bound of $I$ is increased as we add $[0, \infty)$. It can be the case that $I$ is not restricted (if the lower bound of $I$ is contained in $tp_{\text{preds}}$) or that the whole time window is cut out (if $I \cap tp_{\text{preds}} = \varnothing$). But it never happens that a time window $I$ is cut into multiple time windows. Similarly, in Line 22 only the upper bounds of the time windows are restricted. Again, the intervals are not divided into several. $\square$

**Theorem 8.8.** Let $N = (G, v_s, v_t, d, tp, W)$ be an FPTPW network based on the graph $G = (V, A)$ and let $\overleftrightarrow{tp}$ be the time profiles generated by Algorithm 8.1 on input $N$. Define $M = \max_{v \in V}\{\max\{|tp(v)|, |\overrightarrow{tp}(v)|, |\overleftrightarrow{tp}(v)|\}\}$ and let $\Delta = \max\{\Delta^{\text{in}}(G), \Delta^{\text{out}}(G)\}$. Then, Algorithm 8.1 runs on input $N$ in time $\mathcal{O}(\log(\Delta) \cdot M \cdot |A|)$.

*Proof.* We can compute a topological ordering of the graph $G$ in Line 1 of Algorithm 8.1 in time $\mathcal{O}(|V| + |A|) = \mathcal{O}(|A|)$ [see CLRS09, Chapter 22.4].

The running time of the two `for`-loops is essentially the same, so we only consider the first one (Lines 3 to 13). Note that $|V| \in \mathcal{O}(|A|)$ since $G$ is connected.

To unite time profiles of the predecessors in Line 4 we have to perform a $k$-way merge. Merging $|N^{\text{in}}(v_i)| \leq \Delta^{\text{in}}(G)$ time profiles with at most $M$ time windows each (that is, $|N^{\text{in}}(v_i)| \cdot M$ in total) results in a running time of $\mathcal{O}\big(\log(\Delta^{\text{in}}(G)) \cdot |N^{\text{in}}(v_i)| \cdot M\big)$, see [Knu98, Chapter 5.4.1].

In the case $v_i \notin W$ we have to intersect $tp(v_i)$ with $tp_{\text{preds}}$. We can achieve this by iterating over both time profiles simultaneously. This requires a running time of $\mathcal{O}(|tp(v_i)| + |tp_{\text{preds}}|) \leq \mathcal{O}\big(|N^{\text{in}}(v_i)| \cdot M\big)$. We can also deal with case $v_i \in W$ in this way. The only difference is that we only restrict time windows $I \subseteq tp(v_i)$ to the left allowing us to possibly skip certain time windows of $tp_{\text{preds}}$.

In total, we can estimate the running time as

$$\mathcal{O}\left(|V| + |A| + \sum_{v \in V} \big(\log(\Delta^{\text{in}}(G)) \cdot |N^{\text{in}}(v)| \cdot M + \log(\Delta^{\text{out}}(G)) \cdot |N^{\text{out}}(v)| \cdot M\big)\right),$$

which simplifies to $\mathcal{O}(\log(\Delta) \cdot M \cdot |A|)$. $\square$

**Remark 8.9.** As already anticipated in Example 8.6, Algorithm 8.1 is output-sensitive. In addition, the running time also depends on the time profiles $\overrightarrow{tp}(v)$ that are only required internally. Modifying Example 8.6 by setting the time window of $v_{n-1}$ to $[0, 1]$, we get an instance in which both $tp$ and $\overleftrightarrow{tp}$ consist of only a single time window per vertex. In contrast, the number of time windows of $\overrightarrow{tp}$ still grows exponentially during the forward iteration. Thus, for certain instances Algorithm 8.1 has a running time that is exponential both in the instance and in the output size. $\lhd$

**Corollary 8.10.** For an FPTPW network with $W = V$, Algorithm 8.1 has a running time that is polynomial in the input size.

*Proof.* By Lemma 8.7 we get that $M$ from Theorem 8.8 simplifies to $\max_{v \in V} |tp(v)|$ which is polynomial in the input size. $\square$

## Conclusion

We described how to restrict the time profiles of an FPTPW network by removing all "unnecessary" parts. This makes the time profile of each vertex fit together with those of its predecessors and successors, which is why we call this *synchronizing*.

Since the synchronization process might create exponentially many new time windows, this algorithm has an exponential running time in general. Moreover, we showed that this might even be the case if the output is polynomial in the instance size.

For waiting vertices, however, we proved that the number of time windows does not increase during the synchronization process. This led to the consequence that Algorithm 8.1 runs in polynomial time if waiting is allowed at every vertex.

# Chapter 9

# Departure-Duration Functions

*We introduce "departure-duration functions" and use them to formulate a subpath optimality criterion. In particular, this criterion leads to a recursive formulation of these functions, which plays a central role in our algorithms solving* MinFPTPW*. Furthermore, we study the structure of these functions. The most fundamental property that we prove is their piecewise linearity, which is crucial for the usage in our algorithms. A deeper analysis of the structure of the departure-duration functions, especially a classification of their breakpoints, allow us to derive running time bounds on our algorithms for certain instance classes later in this thesis.*

## Assumptions

Throughout this chapter, an FPTPW network $N = (G, v_s, v_t, d, tp, W)$ based on a graph $G = (V, A)$ is always given. The following assumptions apply.

**Only Relevant Vertices** $\rightarrow$ see Assumption 7.7 on Page 79
Every vertex is contained in a $v_s$-$v_t$-path.

**Synchronized Time Profiles** $\rightarrow$ see Assumption 8.1 on Page 89
All time profiles are synchronized.

**No Parallels** $\rightarrow$ see Assumption 7.8 on Page 80
The graph does not contain parallel arcs.

## 9.1 Definition and Recursive Formulation

The departure-duration function of a vertex $v$ maps every point $t \in tp(v)$ to the minimum duration of a valid $v_s$-$v$-path $(P, \tau)$ that satisfies $\tau(v) = t$. Informally speaking, this function maps a departure time to the duration of a "fastest path to get there". It is of interest when we look for the (duration of a) fastest path that departs at a vertex $v$ at a fixed time $t$. Analogously, we define arrival-duration functions that store durations of fastest paths that arrive at a specific vertex at a fixed time. In the following chapters, we mainly consider the departure-duration functions. The main motivation to introduce the arrival-duration functions is to simplify proofs concerning the departure-duration functions.

**Definition 9.1.** For a vertex $v \in V$, the *departure-duration function* $f_v^\tau$ and the *arrival-duration function* $f_v^\sigma$ are defined by

$$\begin{aligned}
f_v^\tau &\colon tp(v) \to \mathbb{R} \\
&\quad t \mapsto \min\{t - \tau(v_s) : (P, \tau) \text{ valid } v_s\text{-}v\text{-path with } \tau(v) = t\} \quad \text{and} \\
f_v^\sigma &\colon tp(v) \to \mathbb{R} \cup \{\infty\} \\
&\quad t \mapsto \min\{t - \tau(v_s) : (P, \tau) \text{ valid } v_s\text{-}v\text{-path with } \sigma(v) = t\}.
\end{aligned}$$

Note that we use the superscripts $\tau$ and $\sigma$ to indicate that these functions represent departure or arrival information, which is in accordance with our notation for arrival intervals, see Definition 7.3 on Page 78.

**Lemma 9.2.** Let $v \in V$. The departure-duration function $f_v^\tau$ and the arrival-duration function $f_v^\sigma$ are well-defined.

*Proof.* Let $t \in tp(v)$. If no valid $v_s$-$v$-path arrives at $v$ at time $t$, the arrival-duration function is well-defined at time $t$ because its codomain contains infinity. Thus, we may assume that the set of valid $v_s$-$v$-paths with $\sigma(v) = t$ is non-empty. By Assumption 8.1 we obtain that also the set of valid $v_s$-$v$-paths with $\tau(v) = t$ is non-empty. Since both sets contain only nonnegative numbers, the corresponding infima are real-valued. That these are in fact attained, and the minima thus exist, is due to the fact that all time profiles are composed of closed intervals only and because the graph is finite. $\square$

Before we go into detail about departure-duration and arrival-duration functions, we look at a small example to gain some intuition.

**Example 9.3.** Let us consider the FPTPW network given in Figure 9.1. Waiting is only allowed at vertex $v_3$, that is, $W = \{v_3\}$. Throughout this thesis, we draw waiting vertices with filled circles whereas we use blank circles for vertices where waiting is forbidden.

At the source $v_s$, the departure-duration and the arrival-duration function is constant 0 since we can depart and arrive at every time within the time profile $[0, 2]$. In fact, it is not surprising that both functions coincide because we prove in Lemma 9.18 on Page 107 that the departure-duration function of a vertex equals the arrival-duration function if waiting is forbidden at this vertex.

In order to arrive at vertex $v_2$ at a time $t \in [1, 3]$, we have to depart at the source exactly one time unit earlier. In particular, these paths are unique and each has a duration of 1.

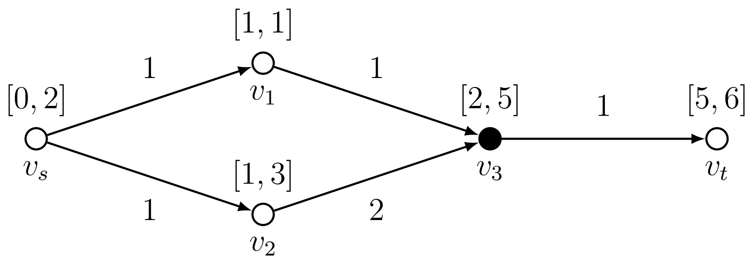
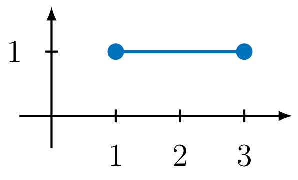
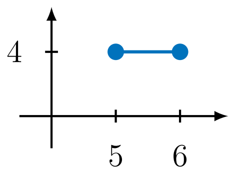


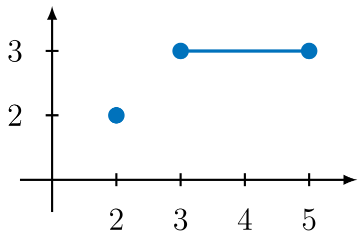
Figure 9.1: The FPTPW network for Examples 9.3 and 9.4. The time profiles are located above the vertices and the durations are next to the arcs. Waiting is only allowed at vertex $v_3$.



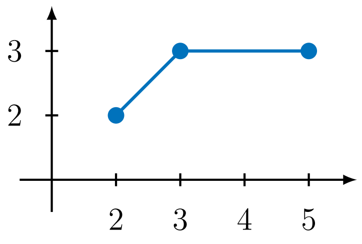
(a) The arrival-duration function $f^{\sigma}_{v_2}$ and the departure-duration function $f^{\tau}_{v_2}$ at vertex $v_2$.



(b) The arrival-duration function $f^{\sigma}_{v_t}$ and the departure-duration function $f^{\tau}_{v_t}$ at vertex $v_t$.



(c) The arrival-duration function $f^{\sigma}_{v_3}$ at vertex $v_3$. In the open interval $(2, 3)$ its value is infinite.



(d) The departure-duration function $f^{\tau}_{v_3}$ at vertex $v_3$.

Figure 9.2: Some selected departure-duration and arrival-duration functions for the FPTPW network from Figure 9.1.

Thus, the corresponding arrival-duration and departure-duration function, visualized in Figure 9.2a, is constant 1.

We can reach vertex $v_3$ by a path via vertex $v_1$ or via $v_2$. In the first case we can only arrive at time 2 because we have to depart at $v_1$ exactly at time 1. This path has duration 2. In the latter case we can arrive at $v_3$ at any time within the interval $[3,5]$ with duration 3 each. Since we cannot arrive within the open interval $(2,3)$, the arrival-duration function is infinite on this interval. It is illustrated in Figure 9.2c.

However, as waiting is permitted at $v_3$, we can arrive at time 2 and wait until every time $t \in (2,3)$ resulting in the departure-duration function for vertex $v_3$ that is drawn in Figure 9.2d. $\triangleleft$

The departure-duration functions store shortest path information for all possible departure times. This allows us to formulate some sort of a subpath optimality criterion in Lemma 9.5. In fact, the following example shows that there is no optimality criterion that is independent of time.

**Example 9.4.** The fastest $v_s$-$v_3$-path in the FPTPW network from Figure 9.1 is unique. It consists of the path $(v_s, v_1, v_3)$ with departure times $\tau(v_s) = 0$, $\tau(v_1) = 1$, and $\tau(v_3) = 2$. However, every fastest $v_s$-$v_t$-path is based on the path $P = (v_s, v_2, v_3, v_t)$. This is because the earliest departure time at the target is 5 and the latest departure time at $v_1$ is 1. Thus, a valid path based on $(v_s, v_1, v_3, v_t)$ has a duration of at least 5. On the other hand, choosing departure times $\tau(v_s) = 1$, $\tau(v_2) = 2$, $\tau(v_3) = 4$, and $\tau(v_t) = 5$ yields a valid path $(P, \tau)$ with duration 4. $\triangleleft$

**Lemma 9.5.** Let $(P, \tau)$ be a $v_s$-$v$-path with $\tau(v) - \tau(v_s) = f_v^\tau(\tau(v))$. Then, also every other vertex $u \in V(P)$ on the path $P$ satisfies $\tau(u) - \tau(v_s) = f_u^\tau(\tau(u))$.

*Proof.* Let $u \in V(P)$. Since the $v_s$-$u$-subpath of $P$ is itself a valid $v_s$-$u$-path, we have $\tau(u) - \tau(v_s) \geq f_u^\tau(\tau(u))$ by the definition of the departure-duration function. If it holds $\tau(u) - \tau(v_s) > f_u^\tau(\tau(u))$, then there is a faster $v_s$-$u$-path departing at $u$ at time $\tau(u)$ (that is, one with a later departure time at the source). Thus, we can improve $(P, \tau)$ by replacing its $v_s$-$u$-subpath with the faster one. Since this does not modify the departure time $\tau(v)$ at $v$, it contradicts the fact that $(P, \tau)$ is a fastest $v_s$-$v$-path arriving at $v$ at time $\tau(v)$. $\square$

The following lemma provides a way to equivalently express the departure-duration functions $f_v^\tau$ recursively. It allows us to derive a couple of properties.

**Lemma 9.6.** The arrival-duration and departure-duration functions at the source are constant 0, that is, $f_{v_s}^\sigma \equiv 0$ and $f_{v_s}^\tau \equiv 0$. For $v \in V \setminus \{v_s\}$ and $t \in tp(v)$ they can be expressed by

$$f_v^\tau(t) = \min\{f_v^\sigma(t') + (t - t') : t' \in I_v^\sigma(t)\} \quad \text{and} \tag{9.1}$$
$$f_v^\sigma(t) = \min\{f_u^\tau(t - d(uv)) + d(uv) : u \in N^{\text{in}}(v) \text{ and } t - d(uv) \in tp(u)\}. \tag{9.2}$$

*Proof.* A valid $v_s$-$v_s$-path consists of exactly the vertex $v_s$ at which we can depart at every point in its time profile. Thus, we have $f_{v_s}^\tau \equiv 0$. Since the arrival time at the first vertex of a path is by definition the same as the departure time, we also have $f_{v_s}^\sigma \equiv 0$.

To prove the recursive formulas (9.1) and (9.2), let $v \in V \setminus \{v_s\}$ and let $t \in tp(v)$.

We start with Equation (9.1). Every valid $v_s$-$v$-path $(P, \tau)$ with $\tau(v) = t$ must satisfy $\sigma(v) \in I_v^\sigma(t)$. Reformulating Definition 9.1 yields

$$\begin{aligned} f_v^\tau(t) &= \min\{t - \tau(v_s) : (P,\tau) \text{ valid } v_s\text{-}v\text{-path with } \tau(v) = t\} \\ &= \min\{t - t' + t' - \tau(v_s) : (P,\tau) \text{ valid } v_s\text{-}v\text{-path with } \sigma(v) = t' \in I_v^\sigma(t)\} \\ &= \min\{f_v^\sigma(t') + (t - t') : t' \in I_v^\sigma(t)\}. \end{aligned}$$

For Equation (9.2) we first note that arriving at $v$ at time $t$ requires to depart at a predecessor $u \in N^{\text{in}}(v)$ exactly at time $t - d(uv)$. For this to be possible, $t - d(uv) \in tp(u)$ must apply. Thus, we get

$$\begin{aligned} f_v^\sigma(t) &= \min\{t - \tau(v_s) : (P,\tau) \text{ valid } v_s\text{-}v\text{-path with } \sigma(v) = t\} \\ &= \min\{t - d(uv) + d(uv) - \tau(v_s) : u \in N^{\text{in}}(v), t - d(uv) \in tp(u), \text{ and} \\ &\qquad\qquad (P,\tau) \text{ valid } v_s\text{-}v\text{-path with } \tau(u) = t - d(uv)\} \\ &= \min\{f_u^\tau(t - d(uv)) + d(uv) : u \in N^{\text{in}}(v) \text{ and } t - d(uv) \in tp(u)\}. \qquad \square \end{aligned}$$

The formulation of Lemma 9.6 allows us to recursively compute all departure-duration functions $f_v^\tau$ for $v \in V$. If we additionally backtrack predecessor information, we are thus able to solve MinFPTPW: compute all departure-duration functions, determine the minimum of the one at the target $f_{v_t}^\tau$, and use the predecessor information to reconstruct the corresponding fastest path. In fact, this approach is our base algorithm for solving MinFPTPW, which we discuss and analyze in detail in Section 10.2. An essential point that allows the efficient representation of the departure-duration functions (which makes this algorithm practically feasible) is their piecewise linearity, see Lemma 9.8.

**Remark 9.7.** Due to the assumption that $G$ has no parallels we can iterate in Equation (9.2) in Lemma 9.6 over all predecessor $u \in N^{\text{in}}(v)$. However, this can also be generalized to graphs that contain parallels by iterating over all arcs entering $v$. $\triangleleft$

## 9.2 Properties

We split the statements about the departure-duration functions into two parts. First, in Section 9.2.1, we consider properties that are independent of the waiting characteristics of the vertices. Second, in Section 9.2.2, we deal with properties that depend on whether waiting is allowed at the respective vertex or not. There, we focus on the case that waiting is permitted.

### 9.2.1 General Properties

This section primarily deals with structural properties of the arrival-duration and the departure-duration functions. In Lemma 9.8 we prove that they are piecewise linear and that every piece has slope 0 or 1. After that, we classify the breakpoints of these functions in Definition 9.10 and Lemma 9.11. For two types of these breakpoints, namely *left bend* and *jump down*, we prove in Lemmas 9.16 and 9.17 that valid paths corresponding to such breakpoints cannot be shifted forward or backward in time, compare with Lemmas 7.18 and 7.19 on Pages 84 and 85.

As we have seen in Example 9.3, the arrival-duration functions might be infinite on some intervals. We regard such a piece as linear and define its slope to be 0. This allows us to formulate and prove the following lemma.

**Lemma 9.8.** For $v \in V$ the functions $f_v^\sigma$ and $f_v^\tau$ are piecewise linear. In addition, each linear piece containing more than one point has either slope 0 or 1.

*Proof.* We prove the claim by induction on the vertices in a topological ordering. For the source $v_s$ we have $f_{v_s}^\sigma \equiv 0 \equiv f_{v_s}^\tau$ which is piecewise linear and all pieces have slope 0.

Let $v \in V \setminus \{v_s\}$ be some vertex and assume the claim holds for all vertices that precede $v$ in a topological ordering. In particular, this implies that the claim holds for all predecessors of $v$.

By Equation (9.2) from Lemma 9.6 the arrival-duration function $f_v^\sigma$ is the lower envelope of the functions $f_u^\tau + d(uv)$ for the predecessors $u \in N^{\text{in}}(v)$. Because there are only finitely many predecessors and their departure-duration functions are piecewise linear, the arrival-duration function $f_v^\sigma$ is also piecewise linear. Moreover, since every piece of $f_v^\sigma$ is either constant infinity or part of a piece in the departure-duration function of some predecessor, each slope is also either 0 or 1.

By Equation (9.1) from Lemma 9.6, the departure-duration function $f_v^\tau$ only depends on the arrival-duration function $f_v^\sigma$. If a fastest path departing at $v$ at time $t \in tp(v)$ requires waiting at $v$, we have $f_v^\tau(t) = f_v^\sigma(t') + (t - t')$ for some arrival time $t' < t$. In this case, $f_v^\tau(t'') = f_v^\sigma(t') + (t'' - t')$ holds for every departure time $t'' \in [t', t]$: if otherwise $f_v^\tau(t'') = f_v^\sigma(t''') + (t'' - t''') < f_v^\sigma(t') + (t'' - t')$, we obtain the contradiction

$$f_v^\tau(t) \leq f_v^\sigma(t''') + (t - t''') < f_v^\sigma(t') + (t - t') = f_v^\tau(t).$$

We get that the departure-duration function $f_v^\tau$ restricted to the interval $[t', t]$ is a linear function of slope 1. Thus, those $t \in tp(v)$ where $f_v^\tau(t) \neq f_v^\sigma(t)$ are contained in linear pieces of slope 1. □

**Notation 9.9.** We call a linear piece of a departure-duration function *singleton piece* if it only consists of a single time point. Otherwise, we call it *horizontal piece* if its slope is 0 and *waiting piece* if its slope is 1.

In the following definition, we categorize the *breakpoints* of $f_v^\tau$, that is, the points at which $f_v^\tau$ is not differentiable. The four types are also visualized in Figure 9.3. Note that a departure-duration function always takes the smaller value at discontinuity points because it is defined as a minimum and since all time windows are closed intervals.

**Definition 9.10** (Breakpoints)**.** Let $v \in V$. An interior point $t \in \operatorname{int}(tp(v))$ in the time profile of vertex $v$ is called

- ▷ a *jump down* if $f_v^\tau(t) < \lim_{t' \nearrow t} f_v^\tau(t')$,
- ▷ a *jump up* if $f_v^\tau(t) < \lim_{t' \searrow t} f_v^\tau(t')$,
- ▷ a *left bend* if a $\delta > 0$ exists such that for all $0 < \varepsilon < \delta$ we have $f_v^\tau(t - \varepsilon) = f_v^\tau(t)$ and $f_v^\tau(t + \varepsilon) = f_v^\tau(t) + \varepsilon$, or
- ▷ a *right bend* if a $\delta > 0$ exists such that for all $0 < \varepsilon < \delta$ we have $f_v^\tau(t - \varepsilon) = f_v^\tau(t) - \varepsilon$ and $f_v^\tau(t + \varepsilon) = f_v^\tau(t)$.

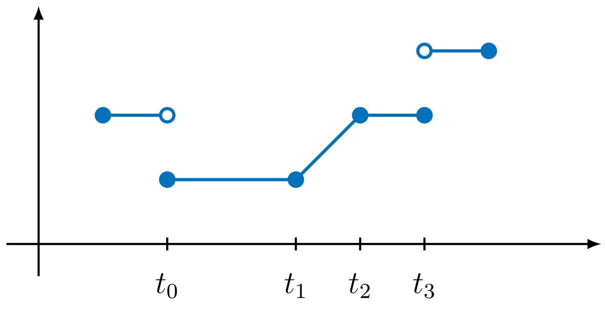


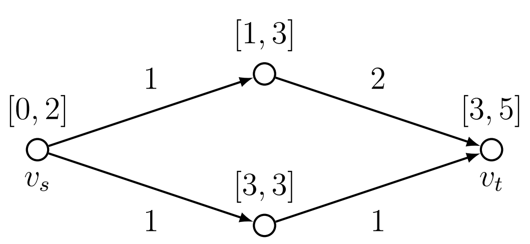
Figure 9.3: Schematic view of the four different types of breakpoints of departure-duration functions, see Definition 9.10. Depicted are a jump down at $t_0$, a left bend at $t_1$, a right bend at $t_2$, and a jump up at $t_3$.

The following lemma shows that all breakpoints of $f_v^\mathcal{T}$ are covered by the four types from Definition 9.10.

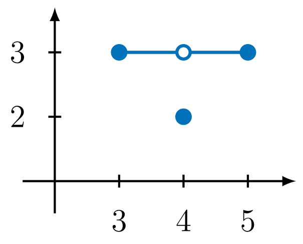
**Lemma 9.11.** Let $v \in V$ and let $t \in \operatorname{int}(tp(v))$ be a point at which $f_v^\mathcal{T}$ is not differentiable. Then $t$ is a jump down, a jump up, a left bend, or a right bend.

*Proof.* By Lemma 9.8 the function $f_v^\mathcal{T}$ is piecewise linear and every piece has slope 0 or 1. The non-differentiable points are exactly the end points of these pieces. If $f_v^\mathcal{T}$ is not continuous at $t$, we have a jump down, a jump up, or both a jump down and a jump up (see also Remark 9.12 for more information on the latter). Otherwise the slope either changes from 0 to 1 or from 1 to 0. In the first case we are exactly in the situation of a left bend and in the second case we have a right bend. □

**Remark 9.12.** A time point $t \in \operatorname{int}(tp(v))$ can be both a jump down and a jump up. However, in this case the point $t$ alone forms a linear piece. It only happens if the fastest $v_s$-$v$-path neither can be shifted forward nor backward in time for some arbitrarily small $\varepsilon > 0$. Such a situation is depicted in Figure 9.4. ◁



(a) FPTPW network



(b) Departure-duration function $f_{v_t}^\mathcal{T}$

Figure 9.4: A point in time can both be a jump down and a jump up as described in Remark 9.12.

To understand the departure-duration functions even more precisely, we connect the pieces with their corresponding paths.

**Lemma 9.13.** Let $v \in V$. Every point in the interior of a waiting (horizontal) piece of $f_v^\mathcal{T}$ corresponds to a fastest $v_s$-$v$-path that does (not) make use of waiting.

*Proof Sketch.* The proof of Lemma 9.8 shows that pieces of slope 1 are only due to waiting. We solely start with pieces of slope 0 at the source. In the transition from departure-duration functions to arrival-duration functions (see Lemma 9.6) we only take lower envelopes of shifted functions. Thereby, the slopes of the pieces are preserved. This also applies for the transition from arrival-duration to departure duration functions at the points where a fastest path does not wait. Otherwise, if a fastest path requires waiting time, this results in a piece of slope 1. □

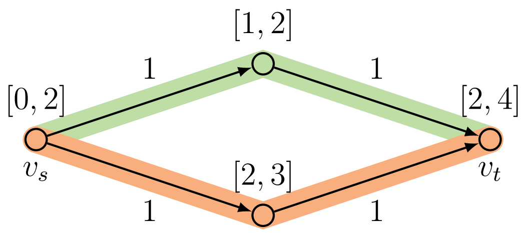
A direct consequence of Lemma 9.13 is that waiting pieces are only induced by vertices at which waiting is allowed, which we state in the following corollary.

**Corollary 9.14.** In the case that waiting is forbidden everywhere, that is, $W = \varnothing$, all departure-duration functions consist of horizontal pieces only.

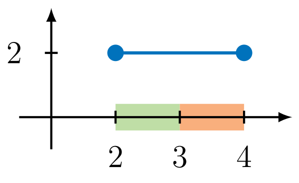
*Proof.* By Lemma 9.13, a point in the interior of a waiting piece would make use of waiting. However, this is not possible as $W = \varnothing$. $\square$

**Example 9.15.** We can associate points on departure-duration functions with valid paths of the specific duration that depart at these points. In doing so, valid paths that are based on the same spatial path of the underlying graph often correspond to points on the same piece of the departure-duration function as their duration only differs in the waiting time.

However, there is no unique correlation between pieces and paths in the underlying graph $G$. On the one hand, multiple paths can correspond to the same piece as shown in Figure 9.5. On the other hand, a path can correspond to multiple pieces of the departure-duration function, see Figure 9.6. $\triangleleft$

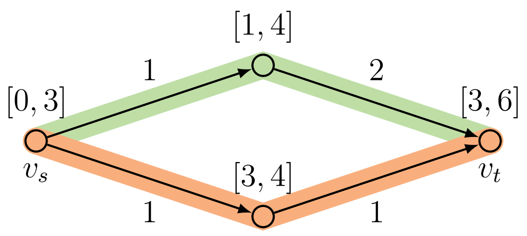


(a) FPTPW network

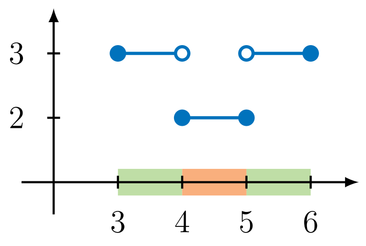


(b) Departure-duration function $f^{\mathcal{T}}_{v_t}$

Figure 9.5: Two paths corresponding to one piece of the departure-duration function. Waiting is forbidden everywhere.



(a) FPTPW network



(b) Departure-duration function $f^{\mathcal{T}}_{v_t}$

Figure 9.6: One path corresponding to two pieces of the departure-duration function. Waiting is forbidden everywhere.

At first glance, the statements of the following two lemmas, the Left Bend Lemma and the Jump Down Lemma, are quite arbitrary. They state that fastest paths according to a left bend or a jump down depart at some vertex at the boundary of a time window. However, these two lemmas will play a central role in bounding the complexity of the departure-duration functions for the case that waiting is allowed everywhere. We will use these statements when we investigate this case in Section 11.2.

**Lemma 9.16** (Left Bend Lemma)**.** Let $v \in V$ be a vertex and let $(P, \tau)$ be a path corresponding to a left bend $t \in tp(v)$ of $f_v^\tau$. Then, there is a vertex $u \in V(P)$ with a time window $[t_0^u, t_1^u] \subseteq tp(u)$ such that $\tau(u) = t_1^u$.

*Proof.* For $w \in V(P)$ let $[t_0^w, t_1^w] \subseteq tp(w)$ be the time window containing $\tau(w)$. Toward a contradiction, suppose that $\tau(w) < t_1^w$ for all $w \in V(P)$.

With choosing $\delta = \min\{t_1^w - \tau(w) : w \in V(P)\} > 0$ we have $[\tau(w), \tau(w) + \delta] \subseteq tp(w)$ for every $w \in V(P)$. Thus, Lemma 7.18 from Page 84 states that the path $(P, \tau + \varepsilon)$ is valid for every $0 \leq \varepsilon \leq \delta$. In particular, this shifting has no effect on the difference between the departure times of consecutive vertices and we obtain

$$f_v^\tau(t + \varepsilon) \leq (\tau(v) + \varepsilon) - (\tau(v_s) + \varepsilon) = \tau(v) - \tau(v_s) = f_v^\tau(t)$$

contradicting the fact that $t$ is a left bend. □

**Lemma 9.17** (Jump Down Lemma)**.** Let $v \in V$ be a vertex and let $(P, \tau)$ be a path corresponding to a jump down $t \in tp(v)$ of $f_v^\tau$. There is a vertex $u \in V(P)$ with a time window $[t_0^u, t_1^u] \subseteq tp(u)$ such that either $\tau(u) = t_0^u$ or $\tau(u) = t_1^u$.

*Proof.* For $w \in V(P)$ let $[t_0^w, t_1^w] \subseteq tp(w)$ be the time window containing $\tau(w)$. Toward a contradiction, suppose that $t_0^w < \tau(w) < t_1^w$ for all $w \in V(P)$.

We first show that the path $(P, \tau)$ does not make use of waiting. If that were the case, we could shift the subpath up to the first waiting vertex by some small $\varepsilon > 0$ to the right (due to $\tau(w) < t_1^w$). This reduces the waiting time by $\varepsilon$ and thus results in a faster path that arrives at time $t$ at vertex $v$.

Since $(P, \tau)$ does not make use of waiting, the arrival times equal the departure times for all vertices in the path. Thus, with $\delta = \min\{\tau(w) - t_0^w : w \in V(P)\} > 0$ we have

$$[\sigma(w) - \delta, \sigma(w)] = [\tau(w) - \delta, \tau(w)] \subseteq tp(w)$$

for every $w \in V(P)$. This allows us to apply Lemma 7.19 from Page 85 and we obtain that $(P, \tau - \varepsilon)$ is a valid path for every $0 \leq \varepsilon \leq \delta$. Analogous to the proof of Lemma 9.16 we have

$$f_v^\tau(t - \varepsilon) \leq (\tau(v) - \varepsilon) - (\tau(v_s) - \varepsilon) = \tau(v) - \tau(v_s) = f_v^\tau(t)$$

contradicting the fact that $t$ is a jump down. □

## 9.2.2 Properties Depending on Waiting Characteristic

We move on to properties of the departure-duration functions that depend on whether waiting is allowed at the corresponding vertex or not. After a first, small observation about vertices where waiting is forbidden we focus on those where it is allowed. The main result is that we can bound the number of linear pieces $\#\mathrm{p}(f_v^\tau)$ of the departure-duration functions $f_v^\tau$ in this case, see Lemma 9.24 and Corollary 9.25.

**Lemma 9.18.** For $v \notin W$ we have $f_v^\sigma = f_v^\tau$.

*Proof.* By Lemma 7.10 on Page 82 it is $\sigma(v) = \tau(v)$ for a valid $v_s$-$v$-path if $v \notin W$. Thus, the claim directly follows from Definition 9.1 on Page 98. $\square$

From now on, in the rest of this chapter, we consider properties of departure-duration functions at vertices that permit waiting. The following lemma states that $f_v^\tau$ has no jump up if $v$ allows waiting.

**Lemma 9.19.** The departure-duration function $f_v^\tau$ of a vertex $v \in W$ at which waiting is allowed does not contain a jump up.

*Proof Sketch.* Let $v \in W$ and assume that $f_v^\tau$ has a jump up at time $t \in \mathrm{int}(tp(v))$. Let $(P, \tau)$ be a fastest $v_s$-$v$-path corresponding to time $t$. We can extend this path by an arbitrarily small amount $\varepsilon$ of waiting at the end. This results in a path departing at $v$ at time $t + \varepsilon$, whose duration is only exactly $\varepsilon$ larger than $d(P, \tau)$. This contradicts the fact that $f_v^\tau$ contains a jump up at $t$. $\square$

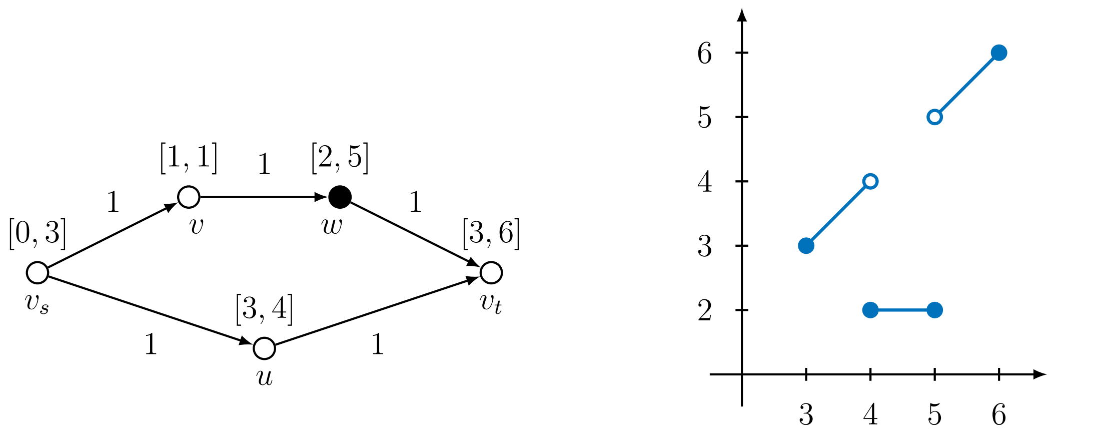
The following lemma states that pieces with slope 1 are not divided within a time window. If for two time points $t_0 < t_1$ of the same time window the value of $f_v^\tau$ differs by $t_1 - t_0$, then $f_v^\tau$ has to be a linear function with slope 1 on the interval $[t_0, t_1]$. In particular, there is no breakpoint of the departure-duration function $f_v^\tau$ within this interval.

**Lemma 9.20.** Let $v \in W$ and let $t_0, t_1 \in tp(v)$ with $t_0 < t_1$ and $[t_0, t_1] \subseteq tp(v)$. If $f_v^\tau(t_1) = f_v^\tau(t_0) + (t_1 - t_0)$, then $f_v^\tau(t) = f_v^\tau(t_0) + (t - t_0)$ for all $t \in [t_0, t_1]$.

*Proof.* Let $(P, \tau)$ be a fastest $v_s$-$v$-path with $\tau(v) = t_0$. Because $v \in W$ and since $[t_0, t_1] \subseteq tp(v)$, we can artificially increase the duration of the path by appending waiting time up to $t_1 - t_0$. Thus, it is $f_v^\tau(t) \leq f_v^\tau(t_0) + (t - t_0)$ for all $t \in [t_0, t_1]$.

Suppose that $f_v^\tau(t) < f_v^\tau(t_0) + (t - t_0)$ for some $t \in [t_0, t_1]$ and let $(P', \tau')$ be a fastest path with $\tau'(v) = t$. Extending this path with $t_1 - t$ waiting at the end results in a path departing at $v$ at time $t_1$ with a duration of $f_v^\tau(t) + (t_1 - t) < f_v^\tau(t_0) + (t_1 - t_0)$, which contradicts the assumption that $f_v^\tau(t_1) = f_v^\tau(t_0) + (t_1 - t_0)$. $\square$

**Example 9.21.** If $v \notin W$, $f_v^\tau$ does not necessarily have to be right-continuous and waiting pieces might be divided. Thus, it is crucial to have the requirement $v \in W$ in Lemmas 9.19 and 9.20. An example that shows this fact is given in Figure 9.7. The only vertex of the FPTPW network that admits waiting is $w$. Since its predecessor $v$ only allows for a single departure time, the departure-duration function of $w$ contains a waiting piece that is propagated to the departure-duration function of the non-waiting vertex $v_t$. This waiting piece in $f_{v_t}^\tau$ is split by a horizontal piece corresponding to the faster but more time-limited path via $u$, see Figure 9.7b. $\triangleleft$



(a) FPTPW network (b) Departure-duration function $f_{v_t}^\tau$

Figure 9.7: If waiting is not allowed at a vertex, the corresponding departure-duration function may contain jumps up and waiting pieces might be divided. See also Example 9.21.

As promised at the start of Section 9.2.2 the main result of this section is to bound the number of linear pieces of departure-duration functions for vertices where waiting is allowed. At this point, let us sketch the general idea to prove this.

Instead of bounding the number of linear pieces we bound the number of breakpoints. By Lemma 9.19 we already know that the departure-duration function of a waiting vertex does not contain a jump up. And since two consecutive breakpoints cannot both be right bends, it is sufficient to bound the number of left bends and jump downs (more details in the proof of Lemma 9.23). To bound these, we associate valid paths to them. By the Left Bend Lemma and the Jump Down Lemma (Lemmas 9.16 and 9.17) these depart at some intermediate vertex at a boundary point of a time window. This way we can associate breakpoints of departure-duration functions with endpoints of time windows (of other vertices). As we have to specify the time windows explicitly when encoding an instance, their number is polynomial in the input size. Thus, it suffices to show that different breakpoints are associated to different endpoints of time windows. To this end we have the following lemma that provides a sufficient criterion to apply Lemma 9.20.

**Lemma 9.22.** Let $v \in W$ and let $t_0, t_1 \in tp(v)$ with $t_0 < t_1$ and $[t_0, t_1] \subseteq tp(v)$. If two fastest $v_s$-$v$-paths $(P_0, \tau_0)$ and $(P_1, \tau_1)$ with $\tau_0(v) = t_0$ and $\tau_1(v) = t_1$, depart at a common vertex $u \in V(P_0) \cap V(P_1)$ at the same time $\tau_0(u) = \tau_1(u)$, then it holds $f_v^\tau(t_1) = f_v^\tau(t_0) + (t_1 - t_0)$.

*Proof.* The condition $\tau_0(u) = \tau_1(u)$ allows interchanging the $v_s$-$u$-subpaths of $(P_0, \tau_0)$ and $(P_1, \tau_1)$. That is, we can replace both the $v_s$-$u$-subpath of $(P_0, \tau_0)$ by $(P_1, \tau_1)|_{v_s u}$ and that of $(P_1, \tau_1)$ by $(P_0, \tau_0)|_{v_s u}$ (see Lemma 7.16 and Definition 7.15). This implies that both paths have the same departure time $\tau_0(v_s) = \tau_1(v_s)$, since otherwise we can improve the duration of one of the two paths (namely the one with the earlier departure time). Finally, the claim follows because $(P_0, \tau_0)$ and $(P_1, \tau_1)$ are fastest paths and, thus, $f_v^\tau(t_1) - f_v^\tau(t_0) = d(P_1, \tau_1) - d(P_0, \tau_0) = t_1 - t_0$. □

**Lemma 9.23.** If $|tp(u)| = 1$ for all $u \in V$, then $\#\mathrm{p}(f_v^\tau) \leq 4 \cdot |V| + 2$ for every vertex $v \in W$ at which waiting is permitted.

*Proof.* Let $v \in W$. As proved in Lemma 9.19 the departure-duration function $f_v^\tau$ does not contain a jump up. By Lemma 9.11 and Remark 9.12 the number of linear pieces of $f_v^\tau$ is exactly one larger than the number of breakpoints. Thus, we prove that the latter is bounded by $4 \cdot |V| + 1$.

Since the piece on the left of a right bend has slope 1 whereas the piece on the right has slope 0, the departure-duration function $f_v^\tau$ does not contain two consecutive right bends. Hence, the number of breakpoints can be estimated above by

$$2 \cdot (\text{number of left bends} + \text{number of jump downs}) + 1$$

and it remains to prove that the total number of left bends and jump downs is bounded by $2 \cdot |V|$. To this end we associate every left bend and jump down $t \in \mathrm{int}(tp(v))$ with a fastest $v_s$-$v$-path departing at $v$ at time $t$.

Lemmas 9.16 and 9.17 now guarantee that every such path uses some vertex $u$ at either the earliest or latest time of its time window (recall that $|tp(u)| = 1$). As there are exactly $2 \cdot |V|$ many of these time points, it is sufficient to show that no two fastest paths use a common one of them. Toward a contradiction, let $(P_0, \tau_0)$ and $(P_1, \tau_1)$ be two fastest paths corresponding to breakpoints $t_0 < t_1$ of the departure-duration function $f_v^\tau$ that depart at some vertex $u$ at the same time (either the earliest or the latest time in the time window of $u$). Now, all requirements of Lemma 9.22 are satisfied. This implies that we can apply Lemma 9.20 and obtain that $f_v^\tau$ has a waiting piece on the interval $[t_0, t_1]$. Hence, $t_1$ has to be a jump down and we get the contradiction

$$f_v^\tau(t_1) < \lim_{t \nearrow t_1} f_v^\tau(t) = f_v^\tau(t_0) + (t_1 - t_0) = f_v^\tau(t_1). \qquad \square$$

**Lemma 9.24.** For $v \in W$, we have $\#\mathrm{p}(f_v^\tau) \leq |tp(v)| \cdot (4 \cdot \sum_{u \in V} |tp(u)| + 2)$.

*Proof Sketch.* Basically, the proof idea is to transform the graph and apply Lemma 9.23. Replacing each vertex $v$ by $|tp(v)|$ many copies, one for each of its time windows, allows us to transform the graph into one with $\sum_{v \in V} |tp(v)|$ many vertices that have single time windows as time profiles each.

Applying Lemma 9.23 to this new instance results in departure-duration functions that have at most $4 \cdot \sum_{v \in V} |tp(v)|$ many linear pieces. If we combine the departure-duration functions of all copies corresponding to one vertex $v$ from the original graph, we obtain the departure-duration function of $v$. This results in the claimed bound. □

**Corollary 9.25.** For $v \in W$, the number of linear pieces of $f_v^\tau$ is polynomially bounded in the input size.

*Proof.* The claim follows directly from Lemma 9.24. □

For $v \in V \setminus W$, the number of linear pieces of $f_v^\tau$ does not have to be polynomially bounded. This might even be the case if all time profiles consist of single time windows each as the following example shows.

**Example 9.26.** We consider an FPTPW network quite similar to that used in the proof of Theorem 7.20 where we show that the synchronizing of time profiles might result in exponentially many time windows. The graph is also a path $(v_0, \ldots, v_{n-1})$ where every arc is doubled. For $i \in \{0, \ldots, n-1\}$, one of the two arcs from $v_i$ to $v_{i+1}$ gets duration 0 and the other gets duration $2^i$ (in contrast to $2^{i+1}$ as in the proof of Theorem 7.20). The time profile of a vertex $v_i$ is set to $tp(v_i) = [0, 2^i]$.

In this case, the time profiles are already synchronized but the departure-duration function $f_{v_i}^\tau$ of a vertex $v_i$ consists of $2^i$ linear segments. As an example, the departure-duration function of vertex $v_2$ is drawn in Figure 9.8. Note that only a single vertex at which waiting is allowed suffices to "smooth out" this step function. ◁

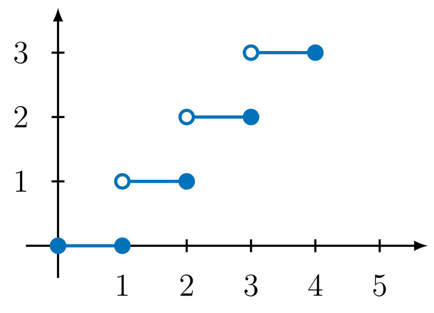


Figure 9.8: Departure-duration function $f_{v_2}^\tau$ for Example 9.26.

## Conclusion

In this chapter, we defined the departure-duration functions, used them to formulate an optimality criterion for fastest paths, and analyzed their structure. Especially, we proved them to be piecewise linear and classified their breakpoints. Furthermore, we derived the Left Bend Lemma and the Jump Down Lemma, which allowed us to associate the breakpoints with endpoints of time windows. As a result, we could polynomially bound the number of linear pieces of the departure-duration functions for vertices at which waiting is allowed.

Moreover, the departure-duration functions are the basic structure of our fastest path algorithms solving MinFPTPW, which we consider in the following chapter.

Chapter 10

# Fastest Path Algorithms

*In this chapter, we present a* MinFPTPW *algorithm that builds on the concept of departure-duration functions that we introduced in the previous chapter. A central task in the algorithm is the computation of lower envelopes, which we therefore first consider as an isolated subproblem. With this preparation, we can then formulate and analyze the algorithm. Thereafter, we bridge the gap to practical applications by discussing variants and extensions of this algorithm that make a difference in practice.*

## Assumptions and Notes

Throughout this chapter, an FPTPW network $N = (G, v_s, v_t, d, tp, W)$ based on a graph $G = (V, A)$ is always given. The following assumptions apply.

**Only Relevant Vertices** → see Assumption 7.7 on Page 79

Every vertex is contained in a $v_s$-$v_t$-path.

**Synchronized Time Profiles** → see Assumption 8.1 on Page 89

All time profiles are synchronized.

**No Parallels** → see Assumption 7.8 on Page 80

The graph does not contain parallel arcs.

In this chapter, we focus on determining the *duration* of a fastest path and sideline the actual computation of a fastest path. The latter can be incorporated into the algorithm without much effort by memorizing predecessor information during its execution. We give more details on this practical extension in Section 10.2.4. A more theoretic approach that justifies the restriction to durations is given in Section 12.1. Specifically, we show there that it is polynomial time equivalent to compute fastest paths and to compute only their durations.

## 10.1 Lower Envelopes

A key task in the MinFPTPW algorithm presented in the following section is to compute lower envelopes of departure-duration functions. The problem of computing lower or upper envelopes occurs not only in combinatorial optimization but it is also important in computer graphics and computational geometry, which is one of the reasons why this problem is already well-investigated. This section aims to summarize the information which is relevant in our situation. We thus focus on lower envelopes of piecewise linear functions in one dimension.

The running time of algorithms computing lower envelopes highly depends on the complexity of this lower envelope function which in turn is related to Davenport-Schinzel sequences. The latter are introduced by Davenport and Schinzel in [DS65] as an approach to transform the problem of how complicated the upper envelope of finitely many continuous functions could become into a purely combinatorial problem. Hart and Sharir prove asymptotically tight bounds on the maximum length of these sequences in [HS86]. It follows from these bounds that the lower (and upper) envelope of $n$ linear segments in the plane consist of at most $\mathcal{O}(n\,\alpha(n))$ segments, where $\alpha$ denotes the inverse Ackermann function. In [WS88], Wiernik and Sharir construct an example where this bound is in the worst case, in fact, tight. A sweep line algorithm to compute upper envelopes of $n$ line segments in time $\mathcal{O}(n\log(n))$ is given in [Her89] by Hershberger. An introduction to Davenport-Schinzel sequences, their analysis, and applications like lower envelope computations can also be found in [AS00].

Our specific problem is to find the lower envelope of $k$ partially defined, piecewise linear functions. We first describe the case for $k = 2$. However, we keep this very brief and refer to [Her89, Section 3] for a more detailed description. Let $f_1$ and $f_2$ be two such functions with $n_1$ and $n_2$ many linear segments. To compute the lower envelope of $f_1$ and $f_2$ we can sweep a line from left to right while always retaining the currently relevant segment of each function. At every breakpoint of one of the functions we can check in constant time whether the following pieces intersect and on which intervals which piece is smaller. This can be turned into an algorithm computing the lower envelope of $f_1$ and $f_2$ in a time that is linear in $n_1$, $n_2$, and the number of linear pieces of the resulting envelope.

As stated above, the lower envelope of $k$ piecewise linear functions with $n$ segments in total has at most $\mathcal{O}(n\,\alpha(n))$ many pieces. Thus, applying the algorithm to determine the lower envelope of two functions successively on the current pointwise minimum and the next function allows us to compute the lower envelope of all $k$ functions in time $\mathcal{O}(kn\,\alpha(n))$.

**Lemma 10.1.** The lower envelope of $k$ piecewise linear functions with $n$ line segments in total can be computed in $\mathcal{O}(kn\,\alpha(n))$ time. □

This algorithm can be improved by a divide-and-conquer strategy where we split the pieces into two equally sized chunks, determine the lower envelopes of those, and finally merge these two functions with the algorithm just like we did before. This approach is described, for example, in [Ata85] or [HS86] and it results in an algorithm running in time $\mathcal{O}(n\,\alpha(n)\log(n))$. With a more sophisticated way to subdivide the linear segments into different chunks, [Her89] improves this approach to an algorithm with a running time of $\mathcal{O}(n\log(n))$.

**Lemma 10.2** ([Her89])**.** The lower envelope of $n$ line segments can be computed in $\mathcal{O}(n\log(n))$ time.

In general, the time bound $\mathcal{O}(n\log(n))$ from Lemma 10.2 is better than $\mathcal{O}(kn\,\alpha(n))$ from Lemma 10.1, especially in the case $k = n$ when every function only consists of a single piece. In our situation, however, the number $k$ is the relatively small indegree of a vertex whereas the number $n$ of linear pieces might become extremely large in practice.

## 10.2 Label Setting Algorithm

In this section, we describe a label setting algorithm for MinFPTPW that is based on the departure-duration functions introduced in Chapter 9. A high-level description of this algorithm is given in Section 10.2.1. Thereafter, in Section 10.2.2, we go into algorithmic details and, in particular, we specify how to represent and compute the departure-duration functions. We derive bounds on the running time and prove the correctness of the algorithm in Section 10.2.3 before we outline how to adapt it in order to compute fastest paths instead of only their durations.

### 10.2.1 Basic Algorithm

To compute the duration of a fastest path in an FPTPW network, the label setting algorithm presented here computes the departure-duration functions of all vertices in the order of a topological ordering. Such a topological ordering of the vertices exists because the graph of every FPTPW network is required to be acyclic. The advantage of this order is that we have already computed the departure-duration functions of all predecessors when we handle a vertex. Thus, we can use the recursive formulation from Lemma 9.6 to compute every departure-duration function. By Assumption 7.7 we start with the source $v_s$ for which we already have $f^{\tau}_{v_s} \equiv 0$. After computing $f^{\tau}_{v_t}$ at the target, the duration of a fastest path equals the minimum of $f^{\tau}_{v_t}$. Furthermore, as we can assume $\sigma(v_t) = \tau(v_t)$ in an optimum solution, there exists a path that arrives at $v_t$ at the time where this minimum is attained. This procedure is summarized in Algorithm 10.1.

**Algorithm 10.1:** Duration of a Fastest Path based on a Topological Ordering

**Input:** FPTPW network $N = (G, v_s, v_t, d, tp, W)$ with $G = (V, A)$, $n = |V|$
**Output:** Duration of a fastest $v_s$-$v_t$-path $(P, \tau)$

```
1 Label the vertices $V$ according to a topological ordering of $G$ with $v_0, \dots, v_{n-1}$.
2 Initialize $f^{\tau}_{v_0} \equiv 0$.
3 for $i = 1, 2, \dots, n-1$ do
4     Determine the departure-duration function $f^{\tau}_{v_i}$.
5 end
6 return $\min_{t \in tp(v_t)} f^{\tau}_{v_t}(t)$
```

## 10.2.2 Algorithmic Details

Algorithm 10.1 sketches the ideas of this label setting algorithm only roughly. In this section, we explain more precisely how we represent the departure-duration functions and how to compute them in Line 4 of the algorithm.

### Representing the Functions

As all functions are piecewise linear functions of a scalar variable, we represent each as a sorted list of its pieces. Every such piece is specified by a tuple $((t_0, y_0), (t_1, y_1))$ containing its two endpoints $(t_0, y_0)$ and $(t_1, y_1)$. These are sufficient to represent departure-duration functions: the values $y_0$ and $y_1$ denote the durations of fastest paths that depart at $t_0$ and $t_1$, respectively. Thus, by the linearity of the piece we have a path of duration $\lambda y_0 + (1 - \lambda) y_1$ that departs at time $\lambda t_0 + (1 - \lambda) t_1$ for all $\lambda \in [0, 1]$.

During the algorithm, all lists containing such pieces are kept sorted lexicographically by the departure times $(t_0, t_1)$. One advantage of this sorting is that we can use a binary search whenever we seek for a piece containing a specific time $t$. In particular, we obtain a value $f^{\tau}_{v}(t)$ in time $\mathcal{O}(\log \#\mathrm{p}(f^{\tau}_{v}))$. Regarding binary search we refer, for example, to [Knu98, Section 6.2.2] or [CLRS09, Chapter 12].

### Computing Departure-Duration Functions

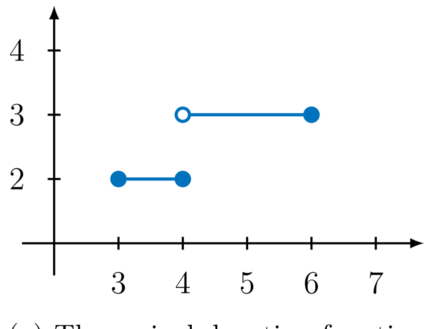
To compute the departure-duration functions in Line 4 of Algorithm 10.1 we use their recursive formulation given by Equation (9.1) in Lemma 9.6. However, these only depend on the arrival-duration functions of the same vertex and not on the departure-duration functions of its predecessors. Thus, we also have to take Equation (9.2) from Lemma 9.6 into consideration.

One way to compute the arrival-duration function $f_v^\sigma$ of a vertex $v$ is given by Equation (9.2) from Lemma 9.6: shift the departure-duration function of each predecessor $u$ by the duration $d(uv)$, compute the pointwise minimum of all these shifted functions, and restrict the result to the time profile of $v$. Shifting a piecewise linear function by $d(uv)$ means that each piece $((t_0, y_0), (t_1, y_1))$ is shifted, which then results in the piece $((t_0 + d(uv), y_0 + d(uv)), (t_1 + d(uv), y_1 + d(uv)))$. To compute the pointwise minimum we have to determine the lower envelope of all these $\sum_{u \in N^{\text{in}}(v)} \#\text{p}(f_u^\tau)$ shifted pieces. The restriction of this lower envelope to the time profile of $v$ can be performed in linear time by iterating all contained pieces once.

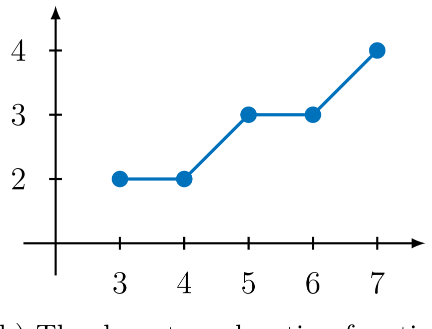
If waiting at the vertex $v$ is forbidden, its departure-duration function equals its arrival-duration function. Otherwise, we have to compute $f_v^\tau$ from $f_v^\sigma$ as specified by Equation (9.1) in Lemma 9.6 on Page 101. That is, we have to add the waiting pieces wherever this is necessary, which is exactly where $f_v^\sigma$ has a jump up. This can also be done in linear time by iterating all pieces of the arrival-duration function once. An example for the addition of such waiting pieces is given in Figure 10.1.

By Lemma 9.24 and Corollary 9.25 we only have bounds on the size of departure-duration functions. Although we can bound the arrival-duration functions with the help of Davenport-Schinzel sequences as described in Section 10.1, we can save their explicit calculation and compute the departure-duration functions directly based on those of the predecessors. At a waiting vertex $v \in W$ we can extend every shifted piece by adding another waiting piece. Thus, we can directly compute its departure-duration function $f_v^\tau$ as the lower envelope of at most $2 \cdot \sum_{u \in N^{\text{in}}(v)} \#\text{p}(f_u^\tau)$ shifted pieces.

So, in order to compute the departure-duration function $f_v^\tau$ at vertex $v$ we must already have the departure-duration functions of all predecessors of $v$ available. This is guaranteed by the fact that we process the vertices in the order of a topological ordering.



(a) The arrival-duration function.



(b) The departure-duration function.

Figure 10.1: Converting an arrival-duration into a departure-duration function for $v \in W$ with $tp(v) = [3, 7]$ by adding two waiting pieces. One at the jump up at 4 and one from the last arrival 6 till the end of the time window.

### 10.2.3 Running Time and Correctness

**Theorem 10.3.** Algorithm 10.1 terminates and it can be implemented to run in time $\mathcal{O}(|A| \cdot \#\mathrm{p}_{\max} \cdot \log(|A| \cdot \#\mathrm{p}_{\max}))$ where $\#\mathrm{p}_{\max} = \max\{\#\mathrm{p}(f_v^\tau) : v \in V\}$ denotes the maximum number of linear pieces among all departure-duration functions.

*Proof.* Initializing the departure-duration function $f_{v_s}^\tau$ at the source in Line 2 requires $\mathcal{O}(|tp(v_s)|) \in \mathcal{O}(\#\mathrm{p}_{\max})$ time, where $|tp(v_s)| = \#\mathrm{p}(f_{v_s}^\tau)$.

As we have already seen in the proof of Theorem 8.8, we can compute the topological ordering of the graph $G$ in Line 1 of Algorithm 10.1 in time $\mathcal{O}(|A|)$ [see CLRS09, Chapter 22.4].

To determine the departure-duration function of a vertex $v$ in Line 4 we have to compute the lower envelope of $\deg^{\mathrm{in}}(v)$ many piecewise linear functions. Each function has at most $\#\mathrm{p}_{\max}$ pieces and, if $v \in W$, we get up to the same amount of additional waiting pieces (compare with the algorithmic details from Section 10.2.2). Thus, we have to compute the lower envelope of at most $2 \cdot \deg^{\mathrm{in}}(v) \cdot \#\mathrm{p}_{\max}$ many linear segments in the plane. By Lemma 10.2 this can be done in time

$$\begin{aligned} &\mathcal{O}\left(\deg^{\mathrm{in}}(v) \cdot \#\mathrm{p}_{\max} \cdot \log(\deg^{\mathrm{in}}(v) \cdot \#\mathrm{p}_{\max})\right) \\ \subseteq\; &\mathcal{O}\left(\deg^{\mathrm{in}}(v) \cdot \#\mathrm{p}_{\max} \cdot \log(|A| \cdot \#\mathrm{p}_{\max})\right). \end{aligned}$$

We have to compute such a lower envelope for every vertex $v \in V$. Thus, the time we spend in the `for`-loop is

$$\begin{aligned} &\mathcal{O}\left(\sum_{v \in V} \deg^{\mathrm{in}}(v) \cdot \#\mathrm{p}_{\max} \cdot \log(|A| \cdot \#\mathrm{p}_{\max})\right) \\ =\; &\mathcal{O}(|A| \cdot \#\mathrm{p}_{\max} \cdot \log(|A| \cdot \#\mathrm{p}_{\max})). \end{aligned}$$

The way we represent the departure-duration functions as described in Section 10.2.2 allows us to find the duration of a fastest path $\min_{t \in tp(v_t)} f_{v_t}^\tau(t)$ in time $\mathcal{O}(\#\mathrm{p}_{\max})$.

In total we get the claimed running time of $\mathcal{O}(|A| \cdot \#\mathrm{p}_{\max} \cdot \log(|A| \cdot \#\mathrm{p}_{\max}))$, and because this running time is bounded, the termination is clear. □

For certain special cases we can limit the running time of Algorithm 10.1 even more precisely by limiting the number $\#\mathrm{p}_{\max}$ of linear pieces, see Chapter 11.

**Theorem 10.4.** Algorithm 10.1 correctly computes the duration of a fastest path.

*Proof.* Algorithm 10.1 terminates by Theorem 10.3. Since the departure-duration functions are computed correctly, the correctness of the algorithm follows from the definition of departure-duration functions (Definition 9.1 on Page 98). □

### 10.2.4 From Durations to Fastest Paths

A slight modification of Algorithm 10.1 allows us to compute a fastest path and not only its duration. To achieve this we can store predecessor information that we can backtrack in order to obtain a fastest path. In the following, we describe how to adapt the algorithm in order to practically obtain a fastest path. An alternative algorithm to determine a fastest path, which simply calls Algorithm 10.1 as a subroutine, is given by Algorithm 12.1 on Page 134. It is more of theoretical interest and proves the polynomial-time equivalence of computing fastest paths and their durations, which is stated in Theorem 12.1.

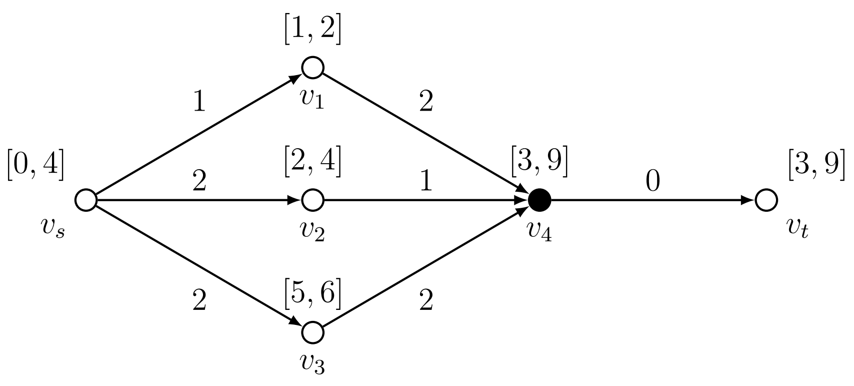
In a standard shortest path computation we can store predecessors next to distance labels if we are also interested in a shortest path and not only in its length [see, for example, AMO93, Section 4.5]. The distance labels correspond to the departure-duration functions in our setting, and we also have to deal with the temporal component in the predecessor information. On the one hand, this means that also the predecessor information is a function in time. On the other hand, this information should not only contain the predecessor but also the time at which we have to depart there.

Formally, we define the predecessor information that we store at a vertex $v$ as the function $\pi_v \colon tp(v) \to V \times \mathbb{R}$. The value $(u, t_u) = \pi_v(t)$ for time $t \in tp(v)$ has the following meaning: there is a $v_s$-$v$-path departing at $v$ at time $t$ with a duration of $f_v^\tau(t)$ such that the predecessor of $v$ on this path is the vertex $u$ at which it departs at time $t_u$. Since the departure-duration functions are piecewise linear (see Lemma 9.8 on Page 102), the predecessor departure times (the second components of $\pi_v$) are also piecewise linear functions. Furthermore, we can attach the predecessors (the first components of $\pi_v$) to the linear pieces of these. This allows a compact representation of the predecessor information that is illustrated in the following example.

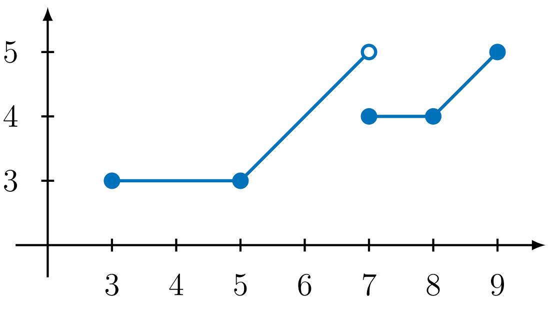
**Example 10.5.** An FPTPW network together with the departure-duration function $f_{v_4}^\tau$ at vertex $v_4$ and the corresponding predecessor information $\pi_{v_4}$ is depicted in Figure 10.2. The FPTPW network depicted in Figure 10.2a contains with the vertex $v_4$ only a single waiting vertex. The departure-duration function is visualized in Figure 10.2b and Figure 10.2c shows the temporal component of $\pi_{v_4}$. Since the information about the predecessor is constant along the pieces, it is written next to these.

In the interval $[3, 4]$, the fastest path is not unique because we can either use the vertex $v_1$ or $v_2$ as a predecessor. Thus, the same applies to the predecessor information: either the piece corresponding to $v_1$ or that corresponding to $v_2$ is contained. To illustrate this, we have exceptionally drawn both pieces in Figure 10.2c.

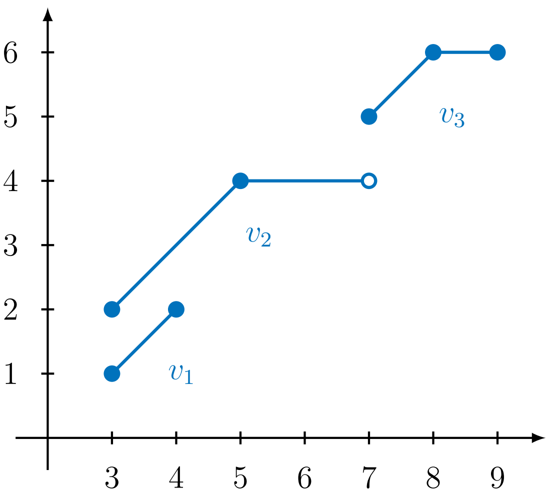
Since the target $v_t$ has the same time profile as $v_4$ and the arc $v_4v_t$ has duration 0, the departure-duration function $f_{v_t}^\tau$ at the target equals $f_{v_4}^\tau$. However, the predecessor information $\pi_{v_t}$ is the identity restricted to the time window $[3, 9]$ enriched with the information that the predecessor is $v_4$. $\triangleleft$



(a) FPTPW network



(b) Departure-duration function $f^{\tau}_{v_4}$



(c) Predecessor information $\pi_{v_4}$

Figure 10.2: The departure-duration function and the corresponding predecessor information at vertex $v_4$ for the given FPTPW network, see also Example 10.5.

We can calculate the predecessor information $\pi_v$ at the same time when determining the departure-duration function $f_v^\tau$ of $v$: instead of only computing the lower envelope of the shifted predecessor pieces we just need to additionally remember where the minima are attained. At the end we can backtrack the predecessor information to obtain a fastest path. If we include these changes into Algorithm 10.1, we obtain Algorithm 10.2.

**Algorithm 10.2:** Fastest Path based on a Topological Ordering

**Input:** FPTPW network $N = (G, v_s, v_t, d, tp, W)$ with $G = (V, A)$, $n = |V|$
**Output:** A fastest $v_s$-$v_t$-path $(P, \tau)$

```
1 Label the vertices $V$ according to a topological ordering of $G$ with $v_0, \dots, v_{n-1}$.
2 Initialize $f_{v_0}^\tau \equiv 0$.
3 Initialize $\pi_{v_0}$ by $t \mapsto (v_0, t)$.
4 for $i = 1, 2, \dots, n-1$ do
5     Determine the departure-duration function $f_{v_i}^\tau$.
6     Determine the predecessor information function $\pi_{v_i}$.
7 end
8 Determine $(P, \tau)$ by backtracking $\pi_v$ starting at $v_t$ at $\arg\min_{t \in tp(v_t)} f_{v_t}^\tau(t)$.
9 return $(P, \tau)$
```

Thanks to the predecessor information functions $\pi_v$, we can backtrack a fastest path in Line 8 of Algorithm 10.2. Assume the minimum of $f_{v_t}^\tau$ is attained at time $t$. We only have to determine $\pi_{v_t}(t)$ to obtain the penultimate vertex $u$ of a fastest path together with the time $t_u$ at which we have to depart there. As long as $u \neq v_s$ we use the predecessor information $\pi_u$ to obtain the preceding vertex on a fastest path together with its departure time.

With these modifications, however, the running time for Algorithm 10.1 from Theorem 10.3 does not simply transfer to Algorithm 10.2: the running time of the latter algorithm also depends on the complexity of the predecessor information functions and on the time to compute these. In particular, it is not clear that complexity-related results on the departure-duration functions carry over to the predecessor information functions.

## 10.3 Improvements

In this section, we outline variants of the fastest path algorithms from the previous section. These variants are mainly of interest from a practical point of view. All improvements have in common that they try to reduce the amount of work instead of speeding it up. Since the ideas are already mentioned in Section 3.5, we will keep this short.

**Label Correcting Algorithm** In Algorithm 10.1 we always compute the departure-duration function of every vertex. However, in this way it can happen that we compute many functions of vertices that are not contained on a fastest path. A first approach to circumvent this issue is to refrain from using the topological ordering. Instead we always process the *most promising* vertex and propagate its departure-duration function to all of its successors.

The most promising vertex in this context is the one with the minimum duration that is not yet propagated to the successors. Whenever we handle a vertex, we update the departure-duration functions of all successors. If this strictly improves the duration $f_u^\tau(t)$ for a successor $u$ at a time $t$, the value $f_u^\tau(t)$ is a duration that vertex $u$ has not yet passed on to its successors. We can stop the algorithm when the minimum duration that has not yet been propagated to the successors (see the start of this paragraph) is attained at the target.

This algorithm does not have to process every vertex, but in general it has to handle vertices multiple times. It might run faster than Algorithm 10.1 if there are many vertices that are only reachable from the source with a duration that is larger than the duration of a fastest $v_s$-$v_t$-path.

**Piece Propagation Algorithm** One drawback of the algorithm variant we just described is that it still propagates the entire departure-duration functions, even if parts of these certainly do not correspond to fastest $v_s$-$v_t$-paths. To address this issue we can propagate only single linear pieces instead of entire departure-duration functions. Therefore, we not only handle the most promising vertex in each iteration but also restrict to propagating the most promising parts of the departure-duration function of this vertex.

With this further adjustment the algorithm essentially computes only those parts of the departure-duration functions that are bounded from above by the duration of a fastest $v_s$-$v_t$-path. In fact, this is not entirely correct as we still propagate pieces instead of single points and because we still pass these values on to succeeding vertices.

**Lower Bounds** We can further improve the practical performance of both variants, the label correcting algorithm as well as the piece propagation algorithm, by refining the choice of the most promising vertex. To this end, we can adapt the A*-algorithm to our setting by computing lower bounds on the remaining durations to the target. We only have to compute a single shortest path tree in a time-independent network, which is relatively fast in comparison to our time-dependent fastest path algorithms. This improvement is particularly helpful if the time profiles fit well in the sense that no large waiting times are needed.

## Conclusion

With Algorithm 10.1 we developed an algorithm to solve MinFPTPW and proved a bound on its running time. We mainly focused on computing the duration of a fastest path but also described how to generalize the algorithm in order to actually compute a fastest path. Finally, we commented on variants of how to improve the algorithm for practical applications.

As the running time of Algorithm 10.1 depends on the complexity of the departure-duration functions, further investigations are necessary in order to figure out whether it runs in time that is polynomial in the input size. We address this question in the following chapter for several special cases.

# Chapter 11

# Special Cases

*This chapter deals with special cases for which we can prove deeper results or that are relevant in practice. First, we restrict the underlying graph and consider the case that it is a path. Second, we investigate the situation in which it is allowed to wait at every vertex. For these first two variants we can solve* MinFPTPW *in polynomial time. Last, we deal with the practically relevant case of integer data.*

## Assumptions

Throughout this chapter an FPTPW network $N = (G, v_s, v_t, d, tp, W)$ based on a graph $G = (V, A)$ is always given. The following assumptions apply.

**Only Relevant Vertices** → see Assumption 7.7 on Page 79

Every vertex is contained in a $v_s$-$v_t$-path.

**Synchronized Time Profiles** → see Assumption 8.1 on Page 89

All time profiles are synchronized.

## 11.1 Path

In this section, we consider MinFPTPW on a path $G = P = (v_0, \dots, v_{n-1})$. By the assumption that all vertices are relevant we have $v_s = v_0$ and $v_t = v_{n-1}$. First, in Section 11.1.1, we start with the case that the time profiles consist of a single time window each. Thereafter, we consider the general case in which the time profiles can be composed of multiple time windows.

### 11.1.1 Single Time Windows

In the case that the time profile of every vertex consists of only a single time window all departure-duration functions have at most two linear pieces. Each such function is a *hinge function* that consists of a horizontal piece with slope 0 followed by a waiting piece with slope 1 (cf. Notation 9.9 on Page 103). However, only one of these two pieces can be there, and, if the time window is a single point, there is only a singleton piece. The recursive formulation of the departure-duration functions from Lemma 9.6 on Page 100 allows us to prove this by induction on the position $i$ of a vertex $v_i$ on the path. In the following, we only sketch the idea of the proof.

At the source $v_s$ the departure-duration function is constant zero and we have as many linear pieces as we have time windows. Hence the departure-duration function at the source consists of a single horizontal piece. Let us now consider a vertex $v_i$ with $i \geq 1$. As $v_{i-1}$ is the only predecessor of $v_i$, the recursive formulation of $f^{\tau}_{v_i}$ from Lemma 9.6 simplifies: we have to shift the departure-duration function $f^{\tau}_{v_{i-1}}$ of $v_{i-1}$ by $d(v_{i-1}v_i)$, restrict it to the time window of $v_i$, and potentially extend it by a waiting piece. Thus, we have that $f^{\tau}_{v_i}$ is a hinge-function as described if the departure-duration function $f^{\tau}_{v_{i-1}}$ of the predecessor has this property.

In particular, this directly implies that we can determine the duration of a fastest path on a path with Algorithm 10.1 from Page 116 in polynomial time.

**Lemma 11.1.** If the underlying graph $G$ of an FPTPW network is a path and every time profile consists of a single time window each, we can find the duration of a fastest path in polynomial time.

*Proof.* If $G$ is a path and every time profile consists of a single time window, every departure-duration function has at most 2 linear pieces. Thus, by Theorem 10.3 from Page 118 we can determine the duration of a fastest path with Algorithm 10.1 in time $\mathcal{O}(|A| \cdot \log |A|)$. □

In this case, however, we can also determine a fastest path and not only its duration in polynomial time. On the one hand, this could be shown by the fact that also the predecessor information in Algorithm 10.2 can be represented by functions with few pieces. On the other hand, the MIP formulation from Section 7.4 simplifies to a linear program that can be solved in polynomial time. In the following, we consider the latter variant.

As the graph $G$ is only a path and because every time profile consists of a single time window, we only have to specify the departure times. This means that we only need the variables $\tau$ from Program (7.1) (see Pages 85 and 86) and neither the $x$-variables, which specify the used vertices and time windows, nor the $y$-variables, which specify the arcs on the path. We denote the single time window of a vertex $v_i$ by $[t^i_0, t^i_1]$. In particular, we have $t_0(v_i) = t_0^i$ and $t_1(v_i) = t_1^i$ with the notation from Section 7.4. Thus, Program (7.1) simplifies for the case of a path $P = (v_0, \dots, v_{n-1})$ with single time windows at the vertices to the following problem.

$$
\begin{aligned}
\min \quad & \tau(v_t) - \tau(v_s) & & \qquad (11.1a)\\
\text{s.t.} \quad & \tau(v_i) \geq t_0^i & \forall\, i = 0, \dots, n-1 & \qquad (11.1b)\\
& \tau(v_i) \leq t_1^i & \forall\, i = 0, \dots, n-1 & \qquad (11.1c)\\
& t_0^i \leq \tau(v_{i-1}) + d(v_{i-1}v_i) & \forall\, i = 1, \dots, n-1 & \qquad (11.1d)\\
& \tau(v_i) \geq \tau(v_{i-1}) + d(v_{i-1}v_i) & \forall\, i = 1, \dots, n-1 & \qquad (11.1e)\\
& \tau(v_i) \leq \tau(v_{i-1}) + d(v_{i-1}v_i) & \forall\, i = 1, \dots, n-1 : v_{i-1} \notin W & \qquad (11.1f)
\end{aligned}
$$

Constraints (11.1b) to (11.1f) from Program (11.1) correspond to Constraints (7.1i) to (7.1m) from Program (7.1). The remaining constraints in the general program are only needed to enforce that the solution corresponds to a $v_s$-$v_t$-path and to involve the time windows.

**Theorem 11.2.** If the underlying graph $G$ of an FPTPW network is a path and every time profile consists of a single time window each, we can find a fastest path in polynomial time.

*Proof.* The formulation from Program (11.1) has only fractional variables $\tau$ making it a linear program. Furthermore, we only need linearly many variables and constraints. The claim follows since we can solve linear programs in polynomial time [see, for example, AP01, Remark 9.11]. □

### 11.1.2 General Time Profiles

We will now move on to more than a single time window at every vertex. Still assuming the underlying graph $G = (v_0, \dots, v_{n-1})$ to be a path we allow general time profiles. Lemma 11.3 shows that we can linearly bound the complexity of the departure-duration functions in this case.

If we allow each time profile to consist of multiple time windows, a departure-duration function restricted to one of the time windows does not have to be a single hinge function. Instead, it can be composed of multiple hinge functions. However, the horizontal pieces of all hinge functions that together form a departure-duration function have the same duration value. This follows from Lemma 9.13 (Page 104) since these points correspond to paths without waiting and their duration is unique on a graph that is a path.

**Lemma 11.3.** Let $N = (G, v_s, v_t, d, tp, W)$ be an FPTPW network based on a graph $G = (v_s = v_0, \ldots, v_{n-1} = v_t)$ that is a path. Then $\#\mathrm{p}(f^\tau_{v_0}) = |tp(v_0)|$ and for $i > 0$

$$\#\mathrm{p}(f^\tau_{v_i}) \leq 2 \cdot (|tp(v_0)| + \sum_{0<j<i} (|tp(v_j)| - 1)) + |tp(v_i)| - 1.$$

*Proof Sketch.* First, we bound the number of horizontal pieces of $f^\tau_{v_i}$. For $f^\tau_{v_0}$ we have exactly $|tp(v_0)|$ horizontal pieces as $f^\tau_{v_0} \equiv 0$, see Lemma 9.6. The departure-duration function $f^\tau_{v_i}$ only has strictly more horizontal pieces than $f^\tau_{v_{i-1}}$ if some of them are split by a hole in the time profile of $v_i$. That is, the maximum number of horizontal pieces of $f^\tau_{v_i}$ is $|tp(v_0)| + \sum_{0<j\leq i}(|tp(v_j)| - 1)$.

Second, a new waiting piece can only be created if we wait after a horizontal piece or if we split a waiting piece by a hole in the time profile. However, splitting a waiting piece creates at most one additional piece whereas splitting a horizontal piece creates up to two additional pieces (one horizontal and potentially a subsequent waiting piece). Hence, we may assume without loss of generality that we only split horizontal pieces.

We are now able to bound the total number of pieces $\#\mathrm{p}(f^\tau_{v_i})$. Starting with $|f^\tau_{v_0}|$ many pieces, every hole in the time profile of some vertex $v_j$ might increase the number of horizontal pieces by one. For $j \leq i$ this affects the departure-duration function of $v_i$. However, only horizontal pieces created by a hole in a preceding vertex $v_j$ (with $j < i$) can be followed by a waiting piece. This results in the claimed bound. □

**Corollary 11.4.** If the underlying graph $G$ of an FPTPW network is a path, we can find the duration of a fastest path in polynomial time.

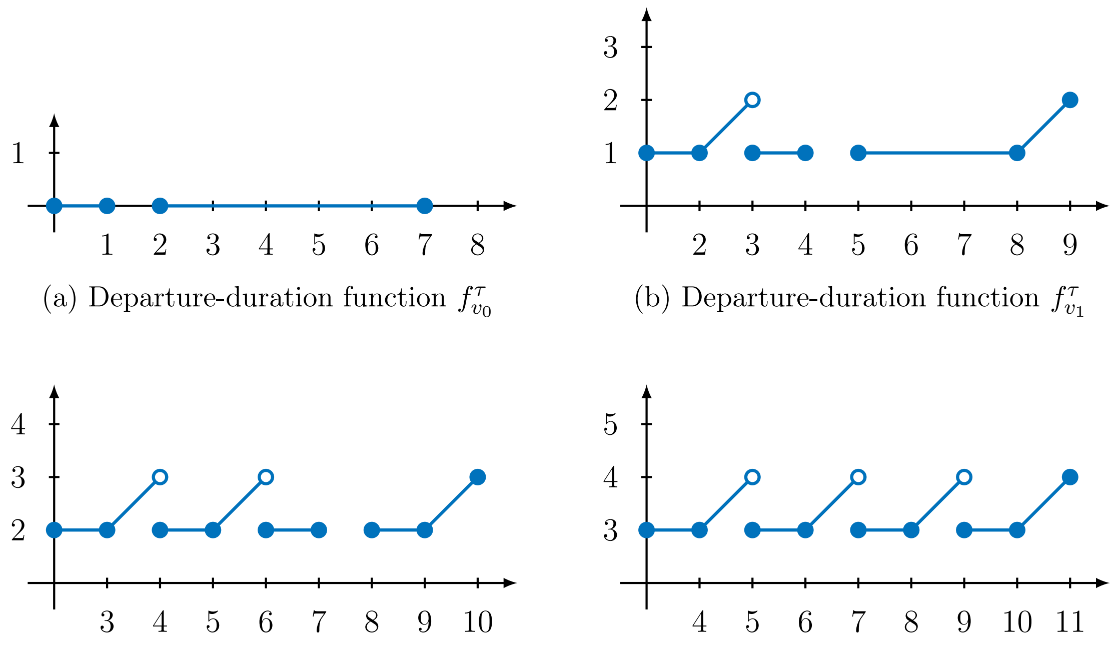
*Proof.* Let $G = (v_0, \ldots, v_{n-1})$. Lemma 11.3 implies $\#\mathrm{p}(f^\tau_{v_i}) \in \mathcal{O}(\sum_{j\leq i} |tp(v_j)|)$ for every vertex, which is linear in the input size. Thus, Theorem 10.3 from Page 118 completes the proof by showing that we can use Algorithm 10.1 to find the duration of a fastest path. □

We finish the special case of a path by providing an example where the bound on the number of linear pieces of departure-duration functions from Lemma 11.3 is tight.

**Example 11.5.** Let $G = (v_0, \ldots, v_{n-1})$ be a path with $v_s = v_0$ and $v_t = v_{n-1}$, let $d \equiv 1$, and let the time profiles be defined by

$$tp(v_i) = \begin{cases} [0, 1] \cup [2, 2n - 1] & \text{if } i = 0, \\ [i, 1 + 3i] \cup [2 + 3i, 2n + i] & \text{if } i < n - 1, \text{ and} \\ [n - 1, 2n + i] & \text{if } i = n - 1. \end{cases}$$

If waiting is allowed at every vertex, the departure-duration function $f^\tau_{v_{n-1}}$ at the target consists of $2n$ pieces. For $n = 4$ the cost functions are shown in Figure 11.1. ◁



(a) Departure-duration function $f^{\mathcal{T}}_{v_0}$

(b) Departure-duration function $f^{\mathcal{T}}_{v_1}$

(c) Departure-duration function $f^{\mathcal{T}}_{v_2}$

(d) Departure-duration function $f^{\mathcal{T}}_{v_3}$

Figure 11.1: Departure-duration functions from Example 11.5 for the case $n = 4$.

## 11.2 Waiting Allowed Everywhere

In this section, we deal with the case that waiting is allowed at every vertex, that is, $W = V$. Basically, we already have all the essential results for this case and only have to put them together.

From Corollary 9.25 (Page 110) we get that all departure-duration functions have only polynomially many pieces. Thus, by Theorem 10.3 from Page 118 we obtain a polynomial running time of Algorithm 10.1 in this case as well.

**Corollary 11.6.** If waiting is allowed at every vertex, that is, if $W = V$, we can find the duration of a fastest path in polynomial time.

In fact, for Corollary 11.6 we do not even need the assumption that all time profiles are synchronized. The reason for this is that by Lemma 8.7 and Corollary 8.10 on Pages 94 and 95 the number of time windows does not increase during the synchronization. Additionally, we can compute the synchronized time profiles with Algorithm 8.1 from Page 91 in polynomial time.

## 11.3 Integer Data

In the project with Deutsche Bahn, which is described in detail in Chapter 3 of this thesis, all time points and durations were integral. We now look at this aspect from a more theoretical point of view. First, in Section 11.3.1, we show that we can restrict to valid paths with integral departure times in this situation. This result allows more efficient representation and handling of the departure-duration functions (see Chapter 9) in fastest path algorithms (see Chapter 10). Thereafter, we draw connections to two different concepts that can be applied if all data is integer. On the one hand, in Section 11.3.2 we consider time-expanded networks and, on the other hand, we look at temporal graphs in Section 11.3.3.

### 11.3.1 Restriction to Integer Departure Times

In the case that all time windows have integer bounds and all durations are integer, the following lemma justifies that we can also restrict to integer departure times.

**Lemma 11.7.** If there is a valid path for an FPTPW network $N$ with only integer data, then there is a fastest path with integer departure times.

*Proof.* Let $i$ be the index of a vertex $v_i$ on the path $P$ at which we wait and an integral time point is contained in the waiting time. Formally, that is an index $i \in \{0, \dots, k\}$ for which $v_i \in W$, $\sigma(v_i) < \tau(v_i)$, and $\lceil \sigma(v_i) \rceil \leq \lfloor \tau(v_i) \rfloor$. If such a vertex does not exist, we can choose $i = -1$ or $i = k + 1$.

We define departure times $\tau'$ for the path $P$ such that $(P, \tau')$ is valid and has a duration that is not larger than the duration of $(P, \tau)$. To this end, let the departure time of a vertex $v_j$ on the path $P$ be

$$\tau'(v_j) = \begin{cases} \lceil \tau(v_j) \rceil & \text{if } j < i \text{ and} \\ \lfloor \tau(v_j) \rfloor & \text{otherwise.} \end{cases}$$

First, we argue that $(P, \tau')$ is valid based on (DT 1) to (DT 4) from Lemma 7.10 on Page 82. Since all durations are integer, the arrival times satisfy $\sigma'(v_j) = \lceil \sigma(v_j) \rceil$ for $j \leq i$ and $\sigma'(v_j) = \lfloor \sigma(v_j) \rfloor$ for $j > i$. And because all bounds of time windows are integer, we have $\lfloor t \rfloor \in I_v$ and $\lceil t \rceil \in I_v$ for every time window $I_v$ in the time profile of a vertex $v$ and a time $t \in I_v$. This shows (DT 1) and (DT 2). As $x \leq y$ implies $\lfloor x \rfloor \leq \lfloor y \rfloor$ and $\lceil x \rceil \leq \lceil y \rceil$, we also get (DT 3) and (DT 4) for all vertices except potentially $v_i$. For this vertex, however, it is $\sigma'(v_i) = \lceil \sigma(v_i) \rceil \leq \lfloor \tau(v_i) \rfloor = \tau'(v_i)$. Thus, the path $(P, \tau')$ is valid.

In the case that such a vertex $v_i$ exists, we have

$$d(P,\tau') = \lfloor \tau(v_k) \rfloor - \lceil \tau(v_0) \rceil \leq \tau(v_k) - \tau(v_0) = d(P,\tau)$$

showing that $P$ with the new departure times $\tau'$ has a duration that is at least as good as the duration with the original departure times $\tau$. If, otherwise, there is no such vertex $v_i$, all waiting times of $(P,\tau)$ are smaller than one second and they does not contain an integral second. By rounding up or down all departure times we eliminate all waiting times and obtain a path $(P,\tau')$ that does not make use of waiting at all. Hence, also in this case the duration of $(P,\tau')$ is at least as good as the duration of $(P,\tau)$. $\square$

### 11.3.2 Time-Expanded Network

If all data is integral, we can transform the time-dependent FPTPW network into a time-expanded graph $G^T = (V^T, A^T)$. Its vertex set $V^T$ consists of a vertex $v^t$ for every $v \in V$ and every integer time point within the time profile $tp(v)$ of $v$. For an arc $uv \in A$ we add all arcs of the form $u^t v^{t+d(uv)}$ to $A^T$ for which both end-vertices are contained in $V^T$. Additionally, for vertices $v \in W$ where waiting is allowed we add arcs $v^t v^{t+1}$ to $A^T$ whenever both end-vertices are in $V^T$. We also add a source and a target that we connect to all vertices corresponding to the source and the target of the FPTPW network, respectively. Formally, we have

$$\begin{aligned} V^T &= \{v^t : v \in V, t \in tp(v)\} \cup \{v_s, v_t\} \quad \text{and} \\ A^T &= \{u^t v^{t+d(uv)} : uv \in A, t \in tp(u), t + d(uv) \in tp(v)\} \\ &\quad \cup \{v^t v^{t+1} : v \in W, [t, t+1] \subseteq tp(v)\} \\ &\quad \cup \{v_s v_s^t : t \in tp(v_s)\} \cup \{v_t^t v_t : t \in tp(v_t)\}. \end{aligned}$$

The duration of an arc $u^t v^{t'}$ in this time-expanded network is defined as the time difference between the times associated with its end-vertices $d(u^t v^{t'}) = t' - t$. A fastest path in the FPTPW network then corresponds to a shortest $v_s$-$v_t$-path in the time-expanded graph $G^T$ where we use the durations as costs.

For the sake of simplicity we assume that all time profiles contain only nonnegative time points, that is, we assume that Assumption 7.9 from Page 80 applies. Denoting the largest time bound of any time window by $M = \max_{v \in V} \max(tp(v))$ allows us to bound the number of vertices $|V^T|$ of the time-expanded network by $|V^T| \leq M \cdot |V| + 2$. Since we need at least $\log(M)$ bits to encode the time window with the largest time bound, the size of the time-expanded graph $G^T$ is pseudo-polynomial in the size of the FPTPW network. For more information about what *pseudo-polynomial* means, we refer to [GJ79, Section 4.2].

Having the time-expanded network graph $G^T$ at hand, the problem MinFPTPW reduces to finding a shortest $v_s$-$v_t$-path with respect to the durations $d$. This can be done by standard shortest path algorithms in time that is polynomial in the size of $G^T$. Thus, we obtain the following result.

**Lemma 11.8.** For an FPTPW network $N$ with only integer data, we can solve MinSFP in pseudo-polynomial time.

This time-expanded approach, however, is most likely not practical in real applications due to the enormous size of the network. A short comparison of time-dependent and time-expanded approaches in the context of timetable information problems is given in [MSWZ07, Section 1.2] where also more related literature is listed. Instead of delving further into this, we want to build a bridge to another topic that is primarily interesting for our problem from a theoretical point of view.

### 11.3.3 Temporal Graphs

Another concept that fits reasonably well is that of temporal graphs. In contrast to the time-expanded approach, the vertices are static and not associated with time points. A temporal graph represents the time by a varying arc set: every arc has an integer time point at which we can start to use it. Hence, we can create the arc set for a temporal graph corresponding to an FPTPW network similarly to the time-expanded graph shown in the previous section. Only the waiting arcs are not necessary, as waiting at vertices is usually allowed in temporal graphs. This prevents the standard temporal graph setting to describe our problem exactly. However, there is current research that deals with restricting the waiting at the vertices, see, for example, [DRSS21] and [CHMZ21].

## Conclusion

If waiting is allowed everywhere or if the graph is a path, we can compute the duration of a fastest path in polynomial time. Moreover, if the path is a graph and the time profile of every vertex consists of a single time window, we can even find the fastest path (and not only its duration) in polynomial time. This result is particularly interesting because we can extend it to all instances for which we can compute the duration of a fastest path in polynomial time. We give more details on this in Section 12.1.

Furthermore, we dealt with instances that only have integer data. The main result for this special case is that we can also restrict the departure times of paths to integer seconds. In addition, we can represent the network as a time-expanded graph in this case, which allows solving MinFPTPW in pseudo-polynomial time.

# Chapter 12

# Complexity

*The main result in this chapter is the polynomial time equivalence of computing fastest paths and of computing only their durations. Furthermore, we specify more accurately when* FPTPW *is polynomially solvable. In the end, we collect and combine the complexity-related results about* FPTPW *to obtain an overall picture.*

## Assumptions

Throughout this chapter, an FPTPW network $N = (G, v_s, v_t, d, tp, W)$ based on a graph $G = (V, A)$ is always given. The following assumption applies.

**Only Relevant Vertices** → see Assumption 7.7 on Page 79

Every vertex is contained in a $v_s$-$v_t$-path.

Recall that we can restrict every directed acyclic graph such that the resulting graph fulfills this assumption and such that every valid path remains valid. Hence, this assumption does not affect the running time statements in this chapter since we only classify whether the problem is polynomially time solvable or not.

## 12.1 Durations versus Fastest Paths

In Section 10.2.4 from Page 119 on, we describe how to incorporate predecessor information in Algorithm 10.1. The result is Algorithm 10.2, which allows us computing fastest paths instead of only their durations. It is an algorithm that is practically of interest, but proving theoretic running time bounds requires further detailed analysis of the predecessor information functions used therein.

Instead, with Algorithm 12.1 we now provide a method that allows us to compute fastest paths in polynomial time if we can determine the duration of a fastest path in polynomial time. From a practical point of view it is probably of little interest but it adds another point to the theoretical consideration of FPTPW and MinFPTPW.

Algorithm 12.1 is based on a similar idea as the algorithm for the Traveling Salesperson Extension that is described in [GJ79, pp. 116–117]. First, we compute the duration of a fastest $v_s$-$v_t$-path in the original graph $G$. Then we successively remove arcs and compute durations of fastest paths in these restricted graphs until we identify a set of arcs that form a fastest path. With the same idea we then proceed in restricting all time profiles to single time windows. In the end, we obtain an instance that is based on a path whose vertices have single time windows as time profiles. Thus, we can solve this instance in polynomial time with Program (11.1) as stated in Theorem 11.2 on Page 127.

**Algorithm 12.1:** Fastest Path based on Fastest Duration

**Input:** FPTPW network $N = (G, v_s, v_t, d, tp, W)$ with $G = (V, A)$, $n = |V|$
**Output:** A fastest $v_s$-$v_t$-path $(P, \tau)$.

```
 1 Compute the duration $d$ of a fastest $v_s$-$v_t$-path in $G$.
   // Restrict the graph to a path
 2 Set $v = v_s$.
 3 while $v \neq v_t$ do
 4     for $vw \in \delta^{\text{out}}(v)$ do
 5         Let $G' = G - (\delta^{\text{out}}(v) \setminus \{vw\})$.
 6         Compute the duration $d'$ of a fastest $v_s$-$v_t$-path in $G'$.
 7         if $d = d'$ then
 8             Let $G = G'$ and set $v = w$.
 9             break // continue with Line 3
10         end
11     end
12 end
   // Restrict the time profiles to single time windows
13 for $v \in G$ do
14     foreach time window $[t_0, t_1] \in tp(v)$ do
15         Let $tp(v) = [t_0, t_1]$.
16         Compute the duration $d'$ of a fastest $v_s$-$v_t$-path.
17         if $d = d'$ then
18             break // continue with Line 13
19         end
20     end
21 end
22 Compute a fastest path $(P, \tau)$ with Program (11.1).
23 return $(P, \tau)$
```

**Theorem 12.1.** It is polynomial-time equivalent to compute a fastest path and to compute only the duration of a fastest path.

*Proof.* Given a fastest path $(P, \tau)$ we can compute the duration of a fastest path in polynomial time by $\tau(v_t) - \tau(v_s)$. The more interesting direction is the reverse, which we prove by showing that Algorithm 12.1 is correct and that it can be implemented to run in polynomial time.

**Correctness** First, in Line 1 we compute the duration $d$ of a fastest $v_s$-$v_t$-path in the complete graph. We then gradually restrict the graph more and more and have to show that $d$ is still the duration of a fastest path after these restrictions.

In Lines 2 to 12 we restrict the graph $G$ to a single $v_s$-$v_t$-path. We end the `for`-loop in Lines 4 to 11 only if the duration $d'$ of a fastest path equals the original duration $d$. Thus, for every vertex $v$ from the outer `while`-loop we have to show that there is always an arc $vw \in \delta^{\text{out}}(v)$ for which this is the case. This can be shown by induction. For the first vertex $v = v_s$ we know that at least one outgoing arc $v_s w$ is contained in a fastest $v_s$-$v_t$-path. Hence, when handling this arc in the `for`-loop, we remove all other arcs leaving $v_s$ and continue with $v = w$. Since $v_s w$ is now the only arc leaving $v_s$, every fastest $v_s$-$v_t$-path has to use an arc leaving $w$.

In Lines 13 to 21 we then restrict the time profiles of the vertices on the path to single time windows. We do this with the same idea as restricting the graph to a path. Analogously, we can show by induction that we always break the inner `for`-loop.

**Running Time** With the `while`-loop in Lines 3 to 12 and the `for`-loop in Lines 4 to 11 we iterate every arc $uv \in A$ of the graph at most once. We can construct $G'$ from $G$ in Line 5 in polynomial time. Additionally, the computation of the duration of a fastest path in Line 6 requires only polynomial time by assumption. Thus, we can restrict the graph to a path (Lines 2 to 12) in polynomial time.

Similarly, we can restrict all remaining time profiles to single time windows in Lines 13 to 21 in polynomial time: The two `for`-loops in this part iterate all time windows of the remaining vertices exactly once. And since we have to encode the time windows separately in the instance, this results in polynomially many iterations. Again, the computation of the duration of a fastest path in Line 16 within every such iteration requires by assumption only polynomial time.

Since the resulting graph is a path and the time profile of every vertex now consists of a single time window, we can compute a fastest path with Program (11.1) as stated in Theorem 11.2 on Page 127. □

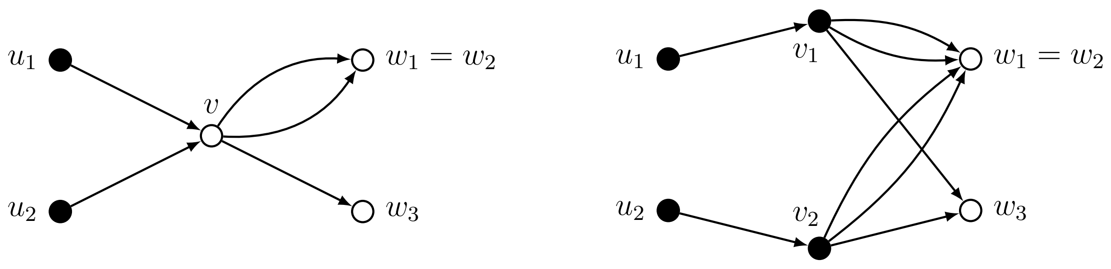
## 12.2 Replacing Vertices

Theorem 7.20 proves MinFPTPW to be **NP**-hard. However, we can solve the problem in polynomial time if waiting is allowed everywhere, see Corollary 11.6. In this section, we further examine the border at which the complexity status changes. The key concept is the replacement of non-waiting vertices. We replace such a vertex $v$ by $\deg^{\text{in}}(v) \cdot \deg^{\text{out}}(v)$ many vertices that all allow waiting. As we do these replacements in the order of a topological ordering, waiting is allowed at all predecessors of a vertex we are replacing. We now describe in detail how to replace the vertices.

For a vertex $v \in V \setminus W$ at which waiting is forbidden let $\delta^{\text{in}}(v) = \{u_1v, \dots, u_kv\}$ and $\delta^{\text{out}}(v) = \{vw_1, \dots, vw_\ell\}$. In particular, we have $k = \deg^{\text{in}}(v)$ and $\ell = \deg^{\text{out}}(v)$. We replace $v$ by $k$ new vertices $v_i$ for $i \in \{1, \dots, k\}$. Accordingly, we replace all arcs incident to vertex $v$. Instead of an arc $u_iv$ we now have the arc $u_iv_i$ that we assign the duration $d(u_iv)$ of the original arc. Every arc $vw_j$ is replaced by $k$ arcs $v_iw_j$, each with a duration of $d(vw_j)$. Waiting at all new vertices is allowed and we set the time profile of a new vertex $v_i$ to

$$tp(v_i) = tp(v) \cap (tp(u_i) + d(u_iv)). \tag{12.1}$$

Note that this construction also works with parallels. If we have, for example, $w_1 = w_2$ we assume that $vw_1$ and $vw_2$ are parallel arcs. The vertex replacement in such a situation is illustrated in Figure 12.1.



(a) The original situation. (b) The new situation.

Figure 12.1: Replacing a vertex $v$.

In the following, we show that such a replacement does not change the duration of a fastest path if waiting is allowed at all predecessors $u_1, \dots, u_k$. More precisely, we show that we can identify valid paths in the original instance with valid paths in the modified instance and vice versa.

By the following lemma we may assume that waiting is allowed at the source and at the target. In fact, for these two vertices, we can always independently choose whether waiting is allowed or not. The reasons are that we can never make use of waiting at the source and that waiting at the target is never useful as it only extends the duration of a path. This allows us to only consider replacements of vertices that are neither the source nor the target.

**Lemma 12.2.** Without loss of generality we may assume $\{v_s, v_t\} \subseteq W$.

*Proof.* For $v_s$ we set $\sigma(v_s) = \tau(v_s)$ anyway, and choosing $\tau(v_t) > \sigma(v_t)$ only increases the duration of a path. □

Note that we can just as well assume $\{v_s, v_t\} \subseteq V \setminus W$ with the same reasoning. Here, however, we are only interested in the statement of Lemma 12.2.

**Lemma 12.3.** Let $v \in V \setminus W$ be a vertex, and let $N'$ be the FPTPW network that results from replacing this vertex as described above. Every valid path $(P, \tau)$ in $N$ corresponds to a valid path in $N'$ with a duration that equals $d(P, \tau)$.

*Proof.* By Lemma 12.2 we may assume $v \notin \{v_s, v_t\}$. Thus, the vertex $v$ has a predecessor $u_i$ on the path $P$. Replacing $v$ on $P$ by $v_i$ and keeping $\tau(v_i) = \tau(v)$ remains valid in $N'$ and does not change the duration of the path. □

**Lemma 12.4.** Let $v \in V \setminus W$ be a vertex whose predecessors all allow waiting, and let $N'$ be the FPTPW network that results from replacing this vertex as described above. Without loss of generality we may assume that no valid $v_s$-$v_t$-path in $N'$ waits at a newly introduced vertex $v_i$.

*Proof.* Suppose that a valid path $(P, \tau)$ in $N'$ waits at a newly introduced vertex $v_i$. The unique predecessor of $v_i$ is $u_i$. The idea is now to move the complete waiting time to $u_i$. At the vertex $v_i$ the path waits during the interval $[\sigma(v_i), \tau(v_i)] \subseteq tp(v_i)$ and Equation (12.1) implies that

$$[\tau(u_i), \tau(v_i) - d(u_i v_i)] = [\sigma(v_i), \tau(v_i)] - d(u_i v_i) \subseteq tp(u_i).$$

Hence, we can set the departure time at $u_i$ to $\tau(v_i) - d(u_i v_i)$. This moves the arrival time at $v_i$ to the departure time there and, thus, removes all waiting time at this vertex. The duration of the modified path remains the same or decreases in the case $u_i = v_s$. □

**Lemma 12.5.** Let $v \in V \setminus W$ be a vertex whose predecessors all allow waiting, and let $N'$ be the FPTPW network that results from replacing this vertex as described above. Every valid path $(P, \tau)$ in $N'$ corresponds to a valid path in $N$ with a duration not larger than $d(P, \tau)$.

*Proof.* If $(P, \tau)$ does not use a newly introduced vertex, it is also a valid path in $N$ and the claim follows. Otherwise, the path uses exactly one new vertex $v_i$. Hence, replacing $v_i$ in $P$ by $v$ transforms $P$ into a path in the original graph $G$. By Lemma 12.4 we can assume that $(P, \tau)$ does not wait at $v_i$. This implies that the departure times remain valid for the original instance $N$. □

We can replace all non-waiting vertices in an FPTPW instance in the order of a topological ordering. Since we can assume $v_s \in W$ by Lemma 12.2, this guarantees that waiting is allowed at all predecessors of the respective vertex to be replaced. Thus, Lemmas 12.3 and 12.5 yield that the duration of a fastest path in the new instance is the same as in the original instance. Furthermore, we can identify the fastest paths in both instances. We obtain the following result.

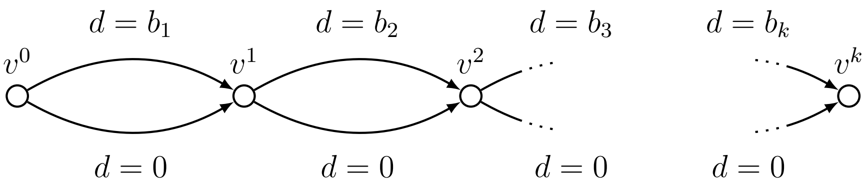
**Corollary 12.6.** We can replace all vertices $v \in V \setminus W$ such that the fastest paths of the resulting instance correspond to fastest paths in the original instance. In addition, the durations of fastest paths coincide. $\square$

The fact that FPTPW is **NP**-hard but polynomially solvable if waiting is allowed everywhere already suggests that the replacement of all non-waiting vertices might blow up the instance exponentially. We now analyze this increase in size and classify instances for which we can polynomially bound it. In the following, we abbreviate the number of vertices and arcs by $n = |V|$ and $m = |A|$, respectively.

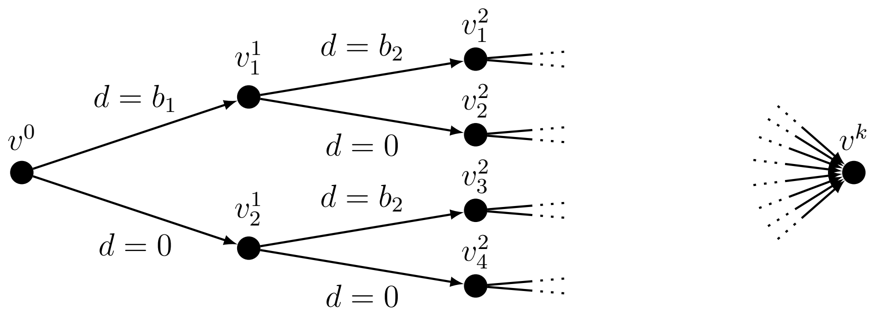
Replacing a single vertex $v$ increases the number of vertices by $\deg^{\text{in}}(v) - 1$ and the number of arcs by $\deg^{\text{out}}(v) \cdot (\deg^{\text{in}}(v) - 1) \in \mathcal{O}(m^2)$. In particular, this increases the indegree of every successor $w \in N^{\text{out}}(v)$ by a factor of at most $\deg^{\text{in}}(v) \in \mathcal{O}(m)$. This is not a problem if waiting at $w$ is allowed. Otherwise, when replacing $w$, this factor propagates amplified by $\deg^{\text{in}}(w)$ to the successors of $w$. The following example illustrates this behavior.

**Example 12.7.** We again consider the FPTPW instance that we construct in the proof of Theorem 7.20 in order to show the **NP**-hardness. This instance represents a SUBSETSUM instance that is given by $B \in \mathbb{N}_{>0}$ and $b_1, \ldots, b_k \in \mathbb{N}_{>0}$. The graph of the FPTPW instance consists of vertices $v^0, \ldots, v^k$ with $v_s = v^0$ and $v_t = v^k$. Compared with the notation from Theorem 7.20 we change from subscript to superscript indices. This allows us to still use subscripts for vertex copies that we create throughout the replacement process. Every two consecutive vertices $v^{i-1}$ and $v^i$ are connected by two parallel arcs, one with duration 0 and the other with duration $b_i$. For the time profiles we have $tp(v_s) = \{0\}$, $tp(v_t) = \{B\}$, and $tp(v^i) = [0, B]$ for all remaining vertices. We repeat Figure 7.1 from Page 87 again in Figure 12.2a.

By Lemma 12.2 we can assume that waiting is allowed at the source and the target. Replacing the remaining vertices $v^1, \ldots, v^{k-1}$ in this order by waiting vertices as described above results in the graph that is drawn in Figure 12.2b. A vertex $v^i$ is replaced by $2^i$ copies $v^i_\ell$ and the graph obtained this way is a binary decision tree. Such a copy $v^i_\ell$ represents a subset $J \subseteq \{1, \ldots, i\}$ and by Equation (12.1) we get that the time profile of $v^i_\ell$ only consists of the time point $\sum_{j \in J} b_j$ if this sum is less or equal than $B$ (otherwise the time profile is empty). $\triangleleft$



(a) The original situation.



(b) The situation after replacing all vertices by waiting vertices.

Figure 12.2: An FPTPW network corresponding to a SubsetSum instance. Replacing all vertices by waiting vertices exponentially increases the graph size.

Example 12.7 demonstrates that the number of required new vertices in the replacement procedure grows exponentially along a path of non-waiting vertices. However, a single waiting vertex after such a path absorbs this increase. If we can bound the number of successive non-waiting vertices on a path, we can also bound the increase from the replacement procedure.

**Lemma 12.8.** Let $k \in \mathbb{N}$. If every path in $G$ with more than $k$ vertices contains at least one waiting vertex $v \in W$, we replacing all vertices $v \in V \setminus W$ results in an increase by a factor of at most $(nm)^k$.

*Proof.* For $v \in V \setminus W$ let $\lambda(v)$ be the maximum length of a path that ends in $v$ and consists only of non-waiting vertices. That is, we define

$$\lambda(v) = \max\{i \geq 1 : (v_1, \dots, v_{i-1}, v) \text{ is a path for } v_1, \dots, v_{i-1} \in V \setminus W\}.$$

We show by an induction on $\lambda(v)$ that replacing a vertex $v \in V \setminus W$ creates at most $n^{\lambda(v)-1} m^{\lambda(v)}$ new vertices and increases the number of arcs by at most this factor. The claim then follows as we have to replace at most $n$ vertices and $\lambda(v) \leq k$ holds for every vertex $v \in V \setminus W$.

For the induction base let $v \in V \setminus W$ with $\lambda(v) = 1$. We have $N^{\text{in}}(v) \subseteq W$ and, thus, at the moment when we replace $v$ we also have $\deg^{\text{in}}(v) \leq m$ and $\deg^{\text{out}}(v) \leq m$. Hence, we introduce at most $m$ new vertices and increase the number of arcs by a factor of at most $m$.

For the induction step let $v \in V \setminus W$ and suppose that the claim holds for all vertices $u$ with $\lambda(u) < \lambda(v)$. By the definition of $\lambda$, every predecessor $u \in N^{\text{in}}(v) \cap W$ satisfies $\lambda(u) < \lambda(v)$. When replacing $v$ we already have replaced all predecessors. For every predecessor $u$ this has increased the number of arcs going out of $u$ to at most $n^{\lambda(v)-2}m^{\lambda(v)}$. As $v$ has at most $n$ predecessors in the original graph, it has at most $n^{\lambda(v)-1}m^{\lambda(v)}$ incoming arcs at the moment when we replace $v$. Thus, we add at most $n^{\lambda(v)-1}m^{\lambda(v)}$ vertices and increase the number of its outgoing arcs by at most this factor. □

**Corollary 12.9.** FPTPW can be solved in polynomial time if $G$ contains at most constantly many successive vertices at which waiting is forbidden.

*Proof.* Follows from Lemma 12.8 together with Corollary 11.6 from Page 129. □

## 12.3 Summary

In this section, we summarize our complexity theoretic results concerning the fastest path with time profiles and waiting problem.

FPTPW is **NP-complete**

- ▷ in general, see Theorem 7.20 on Page 87,
- ▷ if waiting is forbidden everywhere, see Corollary 7.21 on Page 88, and
- ▷ if the graph is series-parallel, see Corollary 7.22 on Page 88.

FPTPW is solvable in **pseudo-polynomial time**

- ▷ if all data is integer, see Lemma 11.8 on Page 132.

FPTPW is solvable in **polynomial time**

- ▷ if the graph is a path, see Corollary 11.4 on Page 128,
- ▷ if waiting is allowed everywhere, see Corollary 11.6 on Page 129, and
- ▷ if the graph has at most constantly many successive vertices at which waiting is forbidden, see Corollary 12.9.

Moreover, it is polynomial time equivalent to compute fastest paths and to compute only their durations, see Theorem 12.1 on Page 135.

## Conclusion

In this chapter, we had two main insights. First, we saw that it is polynomial time equivalent to compute fastest paths and to compute only their durations. Second, we provided a way to replace a non-waiting vertex by polynomially many waiting vertices. The latter allowed us to narrow down the boundary between polynomially solvable and **NP**-hard instances.

# Chapter 13

# Conclusion and Future Research

In Part I, we considered the Almost Disjoint Paths (ADP) and the Separating by Forbidden Pairs (SFP) problem. Based on two exponentially large IP formulations we proved that these problems form a weakly dual pair. In particular, we constructed a family of examples for which the duality gap between ADP and SFP is unbounded. However, these examples only reflect this duality gap in the integrality gap of the ADP formulation, and we are not aware of any instance for which the SFP formulation has an integrality gap. We suspect that this SFP formulation actually has no integrality gap, and first computer-aided verification for small directed acyclic graphs confirm this conjecture. A proof or counter-example for this would fill a small gap in this area, but also more general questions concerning these two problems are of interest. For example, we introduced ADP and SFP only on directed graphs and focused mainly on directed acyclic graphs. However, there is nothing wrong with considering these problems on undirected graphs as well.

We proved that ADP and SFP form a strongly dual pair on directed graphs that have an $s$-$t$-cut with a single outgoing arc. Moreover, both problems are solvable in polynomial time if we restrict them to such graphs (incidentally, this also holds on the problem variants for undirected graphs that have a bridge). However, this is not the case in general. We proved that ADP is **NP**-complete and that SFP is $\mathbf{\Sigma}_2^p$-complete, and both statements still hold if we restrict the problems to directed acyclic graphs. Since both proofs make heavy use of the fact that the graphs are directed, the complexity status of these problems on undirected graphs has not yet been clarified. In addition to examining this point more closely, it would be interesting to find other graph classes on which the duality gap disappears or on which at least one of the problems is significantly easier.

For ADP we presented a dynamic program that allows solving ADP for constant $k$ in polynomial time. Since this dynamic program depends on $\mathcal{O}(m^k 2^{k^2})$ states, where $m$ denotes the number of arcs of the graph, its running time is extremely large. In particular, a solution algorithm based on this dynamic program is not fixed-parameter tractable in $k$. This raises the questions whether ADP is fixed-parameter tractable in $k$ and whether faster algorithms for constant $k$ exist.

In Part II, we considered the Fastest Path with Time Profiles and Waiting (FPTPW) problem. We developed a solution algorithm that propagates departure-duration functions as labels. The structural analysis of these functions gave us insights into the running time of the fastest path algorithm. Particularly interesting at this point is our finding that we can solve FPTPW in polynomial time if all vertices permit waiting. We slightly generalized this result to a constant amount of non-waiting vertices by replacing these by vertices that allow waiting. Instead of restricting the temporal characteristics of an instance by limiting the number of non-waiting vertices we can alternatively simplify the problem by restricting the structure of the graph. In this regard, we proved that FPTPW is polynomial time solvable on a path but it already becomes **NP**-hard if we replace all arcs on such a path by two parallels.

Our recursive formulation of the departure-duration functions and our fastest path algorithm require that the graph is acyclic, which we assume throughout Part II. But FPTPW still makes sense even if we drop this restriction. In this case, other or adapted algorithms are needed. The variants that we describe in Section 10.3 in the context of practical performance should be able to overcome this hurdle. However, deeper analysis of the departure-duration functions and of these algorithms are needed to prove this. In particular, it would be interesting to determine whether the instance classes that we have identified as polynomially solvable remain that way in this more general form.

Since FPTPW is only a simplified version of our practical project, it would still be interesting to consider further problem variants that are closer to this original problem. One possibility would be to simultaneously consider multiple objective functions such as distances or deviations from desired time points. For such a multi-criteria shortest path problem that continuously depend on time, a whole set of exciting questions arise. Specifically, it is already challenging to efficiently store, update, and maintain the efficient paths during a solution algorithm.

Altogether, with ADP, SFP, and FPTPW we introduced three new problems that integrate well into the field of combinatorial optimization. Although very similar to existing problems, they significantly differ from these in complexity and require other solution methods. Therefore, they open up new possibilities and further questions worth considering.

# Appendices

Appendix A

# Notation

*This appendix lists most of the notation used in this thesis. For the sake of clarity and because we use a few symbols in different contexts with different meanings, we split this overview thematically.*

| | |
|---|---|
| $A$ or $A(G)$ | the arc set of a directed graph, see Section 2.2 |
| $\mathbf{APX}$ | the complexity class $\mathbf{APX}$, see Section 2.3 |
| $\deg_G(v)$ | the degree of vertex $v$ in $G$, see Section 2.2 |
| $\deg_G^{\text{in}}(v)$ | the indegree of vertex $v$ in $G$, see Section 2.2 |
| $\deg_G^{\text{out}}(v)$ | the outdegree of vertex $v$ in $G$, see Section 2.2 |
| $E$ or $E(G)$ | the edge set of an undirected graph, see Section 2.2 |
| $G$ | a graph, see Section 2.2 |
| $\text{int}(S)$ | the topological interior of the set $S$, see Section 2.1 |
| $P$ | a path, see Section 2.2 |
| $\mathbf{P}$ | the complexity class $\mathbf{P}$, see Section 2.3 |
| $N_G^{\text{in}}(v)$ | the predecessors of $v$ in $G$, see Section 2.2 |
| $N_G^{\text{out}}(v)$ | the successors of $v$ in $G$, see Section 2.2 |
| $\mathbf{NP}$ | the complexity class $\mathbf{NP}$, see Section 2.3 |
| $V$ or $V(G)$ | the vertex set of a graph, see Section 2.2 |
| $\#\text{p}(f)$ | number of pieces of a piecewise linear function $f$, see Section 2.1 |
| $\alpha(a)$ | the start-vertex of arc $a$, see Section 2.2 |
| $\gamma(e)$ | the end-vertices of an edge $e$, see Section 2.2 |
| $\delta_G^{\text{in}}(v)$ | the arcs of $G$ with end-vertex $v$, see Section 2.2 |
| $\delta_G^{\text{out}}(v)$ | the arcs of $G$ with start-vertex $v$, see Section 2.2 |
| $\Delta^{\text{in}}(G)$ | the maximum indegree of a vertex of $G$, see Section 2.2 |
| $\Delta^{\text{out}}(G)$ | the maximum outdegree of a vertex of $G$, see Section 2.2 |
| $\Delta(G)$ | the maximum degree of a vertex of $G$, see Section 2.2 |
| $\mathbf{\Sigma}_2^p$ | the complexity class $\mathbf{\Sigma}_2^p$, see Section 2.3 |
| $\omega(a)$ | the end-vertex of arc $a$, see Section 2.2 |

Table A.1: Global notation.

| | |
|---|---|
| gad($e$) | a gadget for the **NP**-hardness proof of ADP, see Section 5.3 |
| $H$ | the undirected graph representing an IndSet instance, see Section 5.3 |
| $\mathcal{I}$ | an intersection pattern in the dynamic program, see Theorem 5.2 |
| $m = \lvert E_H \rvert$ | the number of edges of $H$, see Section 5.3 |
| $\mathcal{P}$ | the set of all $s$-$t$-paths, see Chapter 4 |
| $\mathcal{Q}$ | a specific set of almost disjoint paths in $G$, see Assumption 5.7 |
| $s$ | a designated source vertex, see Problem 4.2 |
| $t$ | a designated target vertex, see Problem 4.2 |
| $x$ | the Boolean value of a state in the dynamic program, see Theorem 5.2 |
| $\nu$ | a topological ordering of the graph, see Theorem 5.2 |

Table A.2: Notation concerning ADP.

| | |
|---|---|
| $\mathcal{A}$ | the set of forbidden pairs, see Problem 4.4 |
| $\mathcal{A}_i$ and $\overline{\mathcal{A}_i}$ | optimal separating sets for a variable gadget, see Lemma 6.18 |
| $I$ | an inconsistency of local assignments, see Definition 6.16 |
| $L$ | short for a local $y$-variable assignment $T_{Y(C)}$, see Section 6.2 |
| $\mathcal{P}$ | the set of all $s$-$t$-paths, see Chapter 4 |
| $m$ | the number of clauses in the formula $\varphi$, see Notation 6.15 |
| $n_I$ | number of inconsistencies for formula $\varphi$, see Section 6.2 |
| $n_x = \lvert X \rvert$ | number of $x$-variables occurring in $\varphi$, see Notation 6.10 |
| $n_y = \lvert Y \rvert$ | number of $y$-variables occurring in $\varphi$, see Notation 6.10 |
| $q_i$ | number of occurrences of variable $x_i$ in $\varphi$, see Section 6.2 |
| $s$ | a designated source vertex, see Problem 4.4 |
| $t$ | a designated target vertex, see Problem 4.4 |
| $T$ | an assignment for all variables in $\varphi$, see Notation 6.10 |
| $T_X$ | an assignment for the $x$-variables, see Notation 6.10 |
| $T_Y$ | an assignment for the $y$-variables, see Notation 6.10 |
| $T_{Y(C)}$ | an assignment for the $y$-variables of clause $C$, see Notation 6.15 |
| $X$ | all $x$-variables occurring in $\varphi$, see Notation 6.10 |
| $Y$ | all $y$-variables occurring in $\varphi$, see Notation 6.10 |
| $Y(C)$ | the $y$-variables occurring in clause $C$, see Notation 6.15 |
| $Z = X \cup Y$ | all variables occurring in $\varphi$, see Notation 6.10 |
| $\varphi = \varphi(x, y)$ | a quantified Boolean formula, see Notation 6.10 |

Table A.3: Notation concerning SFP.

| | |
|---|---|
| $A^T$ | the arcs of the time-expanded graph, see Section 11.3 |
| $d(a)$ | the duration of arc $a$, see Definition 7.2 |
| $D$ | the maximum duration for FPTPW, see Problem 7.6 |
| $f_v^\sigma$ | arrival-duration function of vertex $v$, see Definition 9.1 |
| $f_v^\tau$ | departure-duration function of vertex $v$, see Definition 9.1 |
| $G^T$ | the time-expanded graph, see Section 11.3 |
| $N$ | an FPTPW network, see Definition 7.2 |
| $I_v^\sigma(t)$ | the arrival interval at $v$ for departure $t$, see Definition 7.3 |
| $tp(v)$ | the time profile of vertex $v$, see Definition 7.2 |
| $\mathcal{T}$ | the set of all time profiles, see Definition 7.1 |
| $v_s$ | a designated source vertex, see Definition 7.2. |
| $v_t$ | a designated target vertex, see Definition 7.2. |
| $V^T$ | the vertices of the time-expanded graph, see Section 11.3 |
| $W$ | the set of vertices where waiting is allowed, see Definition 7.2. |
| $\alpha$ | the inverse Ackermann function, see Section 10.1 |
| $\pi_v$ | predecessor information for vertex $v$, see Section 10.2 |
| $\sigma(v)$ | the arrival time at vertex $v$, see Definition 7.4 |
| $\tau(v)$ | the departure time at vertex $v$, see Definition 7.4 |

Table A.4: Notation concerning FPTPW.

# Appendix B

# Decision Problems

*This appendix lists the decision problems used within this thesis. The presentation of the problems is inspired from Garey and Johnson's "Guide to the Theory of **NP**-Completeness" [GJ79].*

**Problem B.1** (ADP)**. Almost Disjoint Paths**

**Instance**: A directed graph $G = (V, A)$ with source and target $s, t \in V$ and $k \in \mathbb{Z}_{>0}$.

**Question**: Are there $k$ almost disjoint $s$-$t$-paths in $G$?

*Reference*: Problem 4.2 on Page 26 for the definition, Chapter 5 in general

**Problem B.2** (FPTPW)**. Fastest Path with Time Profiles and Waiting**

**Instance**: A directed acyclic graph $G = (V, A)$, source and target $v_s, v_t \in V$, durations $d\colon A \to \mathbb{R}$, time profiles $tp\colon V \to \mathcal{T}$, and a set of vertices $W \subseteq V$ where waiting is allowed. A maximum duration $D \in \mathbb{R}$.

**Question**: Does a valid path $(P, \tau)$ with $\tau(v_t) - \tau(v_s) \leq D$ exist?

*Reference*: Problem 7.6 on Page 79 for the definition, Part II in general

**Problem B.3** (IndSet)**. Independent Set**

**Instance**: An undirected graph $G = (V, E)$ and a positive integer $k \in \mathbb{Z}_{>0}$.

**Question**: Does $G$ contain an independent set of size $k$?

*Reference*: [GJ79, Problem GT20]

**Problem B.4** (PAFP)**. Path Avoiding Forbidden Pairs**

**Instance**: A directed graph $G = (V, A)$ with source and target $s, t \in V$ and a set $\mathcal{A} \subseteq \binom{A}{2}$ of forbidden pairs.

**Question**: Does an $s$-$t$-path with at most one arc from every forbidden pair exist?

*Reference*: [GMO76] with forbidden pairs of vertices instead of arcs under the name "Impossible Pairs Constrained Path"; Problem 4.6

**Problem B.5** ($\Sigma_2$SAT)**. Quantified Satisfiability with Two Alternations**

**Instance**: A quantified Boolean formula $\varphi(x, y)$ depending on two types of variables.

**Question**: Does an assignment of the $x$-variables exist such that $\varphi(x, y)$ is `true` independently of the assignment of the $y$-variables?

*Reference*: A more general version with $i$ instead of only exactly 2 alternations in [Pap94, Chapter 17] with the abbreviation QSAT$_i$

**Problem B.6** (SFP)**. Separating by Forbidden Pairs**

**Instance**: A directed graph $G = (V, A)$ with source and target $s, t \in V$ and $k \in \mathbb{Z}_{>0}$.

**Question**: Does $k$ arc pairs exist such that every $s$-$t$-path in $G$ contains both arcs of at least one chosen pair?

*Reference*: Problem 4.4 on Page 26 for the definition, Chapter 6 in general

**Problem B.7** (SubsetSum)**. Subset Sum**

**Instance**: Finitely many positive integers $b_1, b_2, \ldots, b_k, B \in \mathbb{Z}_{>0}$.

**Question**: Does $J \subseteq \{1, \ldots, k\}$ with $\sum_{i \in J} b_i = B$ exist?

*Reference*: [Kar72, Section 4] under the name “Knapsack”; [GJ79, Problem SP13]

**Problem B.8** (TSP)**. Traveling Salesperson**

**Instance**: A complete graph $G = (V, E)$ with distances $d\colon E \to \mathbb{Z}_{>0}$ and $B \in \mathbb{Z}_{>0}$.

**Question**: Is there a Hamiltonian cycle in $G$ of length at most $B$?

*Reference*: [GJ79, Problem ND22] under the name “Traveling Salesman”

**Problem B.9** (TSE)**. Traveling Salesperson Extension**

**Instance**: A complete graph $G = (V, E)$ with distances $d\colon E \to \mathbb{Z}_{>0}$, $B \in \mathbb{Z}_{>0}$, and a simple path $P$ in $G$.

**Question**: Can $P$ be extended to a Hamiltonian cycle of length at most $B$?

*Reference*: [GJ79, Chapter 5]

# Bibliography

[AB09] Sanjeev Arora and Boaz Barak. *Computational Complexity: A Modern Approach.* 1st ed. USA: Cambridge University Press, 2009. ISBN: 0521424267.

[ACG+99] Giorgio Ausiello, Pierluigi Crescenzi, Giorgio Gambosi, Viggo Kann, Alberto Marchetti-Spaccamela, and Marco Protasi. *Complexity and Approximation.* Springer Berlin Heidelberg, 1999. DOI: 10.1007/978-3-642-58412-1.

[ADGW10] Ittai Abraham, Daniel Delling, Andrew V. Goldberg, and Renato F. Werneck. "Alternative Routes in Road Networks". In: *Experimental Algorithms.* Springer Berlin Heidelberg, 2010, pp. 23–34. DOI: 10.1007/978-3-642-13193-6_3.

[AEB00] Vedat Akgün, Erhan Erkut, and Rajan Batta. "On finding dissimilar paths". In: *European Journal of Operational Research* 121.2 (2000), pp. 232–246. ISSN: 0377-2217. DOI: 10.1016/S0377-2217(99)00214-3.

[AJKS20] Akanksha Agrawal, Pallavi Jain, Lawqueen Kanesh, and Saket Saurabh. "Parameterized Complexity of Conflict-Free Matchings and Paths". In: *Algorithmica* 82.7 (July 2020), pp. 1939–1965. ISSN: 1432-0541. DOI: 10.1007/s00453-020-00681-y.

[AMO93] Ravindra K. Ahuja, Thomas L. Magnanti, and James B. Orlin. *Network Flows: Theory, Algorithms, and Applications.* Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1993. ISBN: 0-13-617549-X.

[AP01] Dimitris Alevras and Manfred W. Padberg. *Linear Optimization and Extensions.* 1st ed. Springer-Verlag Berlin Heidelberg, 2001. ISBN: 978-3-540-41744-6. DOI: 10.1007/978-3-642-56628-8.

[AS00] Pankaj K. Agarwal and Micha Sharir. "Davenport–Schinzel Sequences and Their Geometric Application". In: *Handbook of Computational Geometry.* Ed. by Jörg-Rüdiger Sack and Jorge Urrutia. Amsterdam: North-Holland, 2000. Chap. 1, pp. 1–47. ISBN: 978-0-444-82537-7. DOI: 10.1016/B978-044482537-7/50002-4.

[Ata85] Mikhail J. Atallah. "Some dynamic computational geometry problems". In: *Computers & Mathematics With Applications* 11 (1985), pp. 1171–1181.

[BBB+15] Marco Blanco, Ralf Borndörfer, Michael Brückner, Nam Dũng Hoàng, and Thomas Schlechte. "On the Path Avoiding Forbidden Pairs Polytope". In: *Electronic Notes in Discrete Mathematics* 50 (2015), pp. 343–348. ISSN: 1571-0653. DOI: 10.1016/j.endm.2015.07.057.

[BBK22] Oliver Bachtler, Tim Bergner, and Sven O. Krumke. *Almost Disjoint Paths and Separating by Forbidden Pairs.* 2022. DOI: 10.48550/arXiv.2202.10090.

[BDGS11] Roland Bader, Jonathan Dees, Robert Geisberger, and Peter Sanders. "Alternative Route Graphs in Road Networks". In: *Theory and Practice of Algorithms in (Computer) Systems.* Springer Berlin Heidelberg, 2011, pp. 21–32. DOI: 10.1007/978-3-642-19754-3_5.

[BDR21] Alexis Bretin, Guy Desaulniers, and Louis-Martin Rousseau. "The traveling salesman problem with time windows in postal services". In: *Journal of the Operational Research Society* 72.2 (2021), pp. 383–397. DOI: 10.1080/01605682.2019.1678403.

[BF95] Piotr Berman and Toshihiro Fujito. "On approximation properties of the Independent set problem for degree 3 graphs". In: *Lecture Notes in Computer Science.* Springer Berlin Heidelberg, 1995, pp. 449–460. DOI: 10.1007/3-540-60220-8_84.

[BK09] Robert L. Burdett and Erhan Kozan. "Techniques for inserting additional trains into existing timetables". In: *Transportation Research Part B: Methodological* 43.8 (2009), pp. 821–836. ISSN: 0191-2615. DOI: 10.1016/j.trb.2009.02.005.

[BK17] Kristof Berczi and Yusuke Kobayashi. "The Directed Disjoint Shortest Paths Problem". en. In: *25th Annual European Symposium on Algorithms (ESA 2017).* Ed. by Kirk Pruhs and Christian Sohler. Vol. 87. Leibniz International Proceedings in Informatics (LIPIcs). Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik GmbH, Wadern/Saarbruecken, Germany, 2017, 13:1–13:13. DOI: 10.4230/LIPIcs.ESA.2017.13.

[BKS+16] Ralf Borndörfer, Torsten Klug, Thomas Schlechte, Armin Fügenschuh, Thilo Schang, and Hanno Schülldorf. "The Freight Train Routing Problem for Congested Railway Networks with Mixed Traffic". In: *Transportation Science* 50.2 (2016), pp. 408–423. DOI: 10.1287/trsc.2015.0656.

[CBG+18] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, Ulf Leser, and David B. Blumenthal. "Finding k-dissimilar paths with minimum collective length". In: *Proceedings of the 26th ACM SIGSPATIAL International Conference on Advances in Geographic Information Systems.* ACM, Nov. 2018, pp. 404–407. DOI: 10.1145/3274895.3274903.

[CBG+20] Theodoros Chondrogiannis, Panagiotis Bouros, Johann Gamper, Ulf Leser, and David B. Blumenthal. "Finding k-shortest paths with limited overlap". In: *The VLDB Journal* 29.5 (Feb. 2020), pp. 1023–1047. DOI: 10.1007/s00778-020-00604-x.

[CCPS98] William J. Cook, William H. Cunningham, William R. Pulleyblank, and Alexander Schrijver. *Combinatorial Optimization.* USA: John Wiley & Sons, Inc., 1998. ISBN: 047155894X.

[CCT10] Valentina Cacchiani, Alberto Caprara, and Paolo Toth. "Scheduling extra freight trains on railway networks". In: *Transportation Research Part B: Methodological* 44.2 (2010), pp. 215–231. ISSN: 0191-2615. DOI: 10.1016/j.trb.2009.07.007.

[CHMZ21] Arnaud Casteigts, Anne-Sophie Himmel, Hendrik Molter, and Philipp Zschoche. "Finding Temporal Paths Under Waiting Time Constraints". In: *Algorithmica* 83.9 (2021), pp. 2754–2802. DOI: 10.1007/s00453-021-00831-w.

[CKT+00] Ting Chen, Ming-Yang Kao, Matthew Tepel, John Rush, and George M. Church. “A Dynamic Programming Approach to de Novo Peptide Sequencing via Tandem Mass Spectrometry”. In: *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*. SODA ’00. San Francisco, California, USA: Society for Industrial and Applied Mathematics, 2000, pp. 389–398. ISBN: 0898714532. DOI: `10.1089/10665270152530872`.

[CL16] Alexis Cornet and Christian Laforest. *Note: Regular languages with no conflicts (forbidden pairs) are regular but have exponential size DFA*. Research Report. LIMOS (UMR CNRS 6158), université Clermont Auvergne, France, Nov. 2016.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. 3rd. The MIT Press, 2009.

[DF57] George B. Dantzig and Delbert R. Fulkerson. “On the Max-Flow Min-Cut Theorem of Networks”. In: *Linear Inequalities and Related Systems*. Princeton University Press, Dec. 1957, pp. 215–222. DOI: `10.1515/9781400881987-013`.

[DGS05] Paolo Dell’Olmo, Monica Gentili, and Andrea Scozzari. “On finding dissimilar Pareto-optimal paths”. In: *European Journal of Operational Research* 162.1 (Apr. 2005), pp. 70–82. DOI: `10.1016/j.ejor.2003.10.033`.

[Die00] Reinhard Diestel. *Graph Theory*. 2nd ed. Vol. 173. Graduate Texts in Mathematics. Springer-Verlag New York, 2000. ISBN: 0-387-95014-1.

[Dij59] Edsger W. Dijkstra. “A note on two problems in connexion with graphs”. In: *Numerische Mathematik* 1.1 (Dec. 1959), pp. 269–271. DOI: `10.1007/BF01386390`.

[DLS98] Guy Desaulniers, June Lavigne, and François Soumis. “Multi-depot vehicle scheduling problems with time windows and waiting costs”. In: *European Journal of Operational Research* 111.3 (1998), pp. 479–494. ISSN: 0377-2217. DOI: `10.1016/S0377-2217(97)00363-9`.

[DRSS21] Umesh Sandeep Danda, G. Ramakrishna, Jens M. Schmidt, and M. Srikanth. “On Short Fastest Paths in Temporal Graphs”. In: *WALCOM: Algorithms and Computation*. Ed. by Ryuhei Uehara, Seok-Hee Hong, and Subhas C. Nandy. Cham: Springer International Publishing, 2021, pp. 40–51. ISBN: 978-3-030-68211-8. DOI: `10.1007/978-3-030-68211-8_4`.

[DS65] Harold Davenport and Andrzej Schinzel. “A Combinatorial Problem Connected with Differential Equations”. In: *American Journal of Mathematics* 87 (1965), p. 684. DOI: `10.2307/2373068`.

[DS88] Martin Desrochers and Francois Soumis. “A Generalized Permanent Labelling Algorithm For The Shortest Path Problem With Time Windows”. In: *INFOR: Information Systems and Operational Research* 26.3 (1988), pp. 191–212. DOI: `10.1080/03155986.1988.11732063`.

[DSD84] Jacques Desrosiers, François Soumis, and Martin Desrochers. “Routing with time windows by column generation”. In: *Networks* 14.4 (1984), pp. 545–565. DOI: `10.1002/net.3230140406`.

[DV00] Guy Desaulniers and Daniel Villeneuve. "The Shortest Path Problem with Time Windows and Linear Waiting Costs". In: *Transportation Science* 34.3 (2000), pp. 312–319. DOI: 10.1287/trsc.34.3.312.12298.

[Ehr05] Matthias Ehrgott. *Multicriteria Optimization.* Berlin, Heidelberg: Springer-Verlag, 2005. ISBN: 3540213988. DOI: 10.1007/3-540-27659-9.

[Eil98] Tali Eilam-Tzoreff. "The disjoint shortest paths problem". In: *Discrete Applied Mathematics* 85.2 (June 1998), pp. 113–138. DOI: 10.1016/S0166-218X(97)00121-2.

[EIS76] Shimon Even, Alon Itai, and Adi Shamir. "On the Complexity of Timetable and Multicommodity Flow Problems". In: *SIAM J. Computing* 5 (Dec. 1976), pp. 691–703. DOI: 10.1137/0205048.

[FGN09] Holger Flier, Thomas Graffagnino, and Marc Nunkesser. "Scheduling Additional Trains on Dense Corridors". In: *Experimental Algorithms.* Ed. by Jan Vahrenhold. Springer Berlin Heidelberg, 2009, pp. 149–160. ISBN: 978-3-642-02011-7. DOI: 10.1007/978-3-642-02011-7_15.

[FHW80] Steven Fortune, John Hopcroft, and James Wyllie. "The directed subgraph homeomorphism problem". In: *Theoretical Computer Science* 10.2 (1980), pp. 111–121. ISSN: 0304-3975. DOI: 10.1016/0304-3975(80)90009-2.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* New York, NY, USA: W. H. Freeman & Co., 1979. ISBN: 0716710447.

[GKT51] David Gale, Harold W. Kuhn, and Albert W. Tucker. "Linear programming and the theory of games". In: *Activity analysis of production and allocation* 13 (1951), pp. 317–335.

[GLS88] Martin Grötschel, Laszlo Lovasz, and Alexander Schrijver. *Geometric Algorithms and Combinatorial Optimization.* 2nd ed. Springer, Berlin, Heidelberg, 1988. ISBN: 978-3-540-56740-0. DOI: 10.1007/978-3-642-78240-4.

[GMO76] Harold N. Gabow, Shachindra N. Maheshwari, and Leon J. Osterweil. "On Two Problems in the Generation of Program Test Paths". In: *IEEE Transactions on Software Engineering* SE-2.3 (Sept. 1976), pp. 227–231. ISSN: 0098-5589. DOI: 10.1109/TSE.1976.233819.

[GWON12] Peter Großmann, Markus Weiss, Jens Opitz, and Karl Nachtigall. "Automated Generation and Optimization of Public Railway and Rail Freight Transport Time Tables". In: *Machines Technologies Materials.* Vol. 6. 5. 2012, pp. 23–26.

[Haa19] Ronald de Haan. *Parameterized Complexity in the Polynomial Hierarchy: Extending Parameterized Complexity Theory to Higher Levels of the Hierarchy.* 1st ed. Lecture Notes in Computer Science. Springer, Berlin, Heidelberg, 2019. ISBN: 978-3-662-60670-4. DOI: 10.1007/978-3-662-60670-4.

[HÁN20] Rebecca Haehn, Erika Ábrahám, and Nils Nießen. "Freight Train Scheduling in Railway Systems". In: *Measurement, Modelling and Evaluation of Computing Systems.* Ed. by Holger Hermanns. Cham: Springer International Publishing, 2020, pp. 225–241. ISBN: 978-3-030-43024-5. DOI: 10.1007/978-3-030-43024-5_14.

[Her89] John Hershberger. “Finding the upper envelope of n line segments in O(n log n) time”. In: *Information Processing Letters* 33.4 (1989), pp. 169–174. ISSN: 0020-0190. DOI: 10.1016/0020-0190(89)90136-1.

[HKKM10] Mohammad Taghi Hajiaghayi, Rohit Khandekar, Guy Kortsarz, and Julián Mestre. “The Checkpoint Problem”. In: *Approximation, Randomization, and Combinatorial Optimization. Algorithms and Techniques.* Ed. by Maria Serna, Ronen Shaltiel, Klaus Jansen, and José Rolim. Springer Berlin Heidelberg, 2010, pp. 219–231. ISBN: 978-3-642-15369-3.

[HNR68] Peter E. Hart, Nils J. Nilsson, and Bertram Raphael. “A Formal Basis for the Heuristic Determination of Minimum Cost Paths”. In: *IEEE Transactions on Systems Science and Cybernetics* 4.2 (July 1968), pp. 100–107. ISSN: 0536-1567. DOI: 10.1109/TSSC.1968.300136.

[HP74] Jonathan Halpern and I. Priess. “Shortest path with time constraints on movement and parking”. In: *Networks* 4.3 (1974), pp. 241–253. DOI: 10.1002/net.3230040304.

[HS86] Sergiu Hart and Micha Sharir. “Nonlinearity of davenport–Schinzel sequences and of generalized path compression schemes”. In: *Combinatorica* 6.2 (June 1986), pp. 151–177. ISSN: 1439-6912. DOI: 10.1007/BF02579170.

[HZ80] Gabriel Y. Handler and Israel Zang. “A dual algorithm for the constrained shortest path problem”. In: *Networks* 10.4 (1980), pp. 293–309. DOI: 10.1002/net.3230100403.

[JKPK09] Yeon-Jeong Jeong, Tschangho John Kim, Chang-Ho Park, and Dong-Kyu Kim. “A dissimilar alternative paths-search algorithm for navigation services: A heuristic approach”. In: *KSCE Journal of Civil Engineering* 14.1 (Sept. 2009), pp. 41–49. DOI: 10.1007/s12205-010-0041-8.

[Kar72] Richard M. Karp. “Reducibility Among Combinatorial Problems”. In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA.* 1972, pp. 85–103. DOI: 10.1007/978-1-4684-2001-2_9.

[KKR12] Ken-ichi Kawarabayashi, Yusuke Kobayashi, and Bruce Reed. “The disjoint paths problem in quadratic time”. In: *Journal of Combinatorial Theory, Series B* 102.2 (Mar. 2012), pp. 424–435. DOI: 10.1016/j.jctb.2011.07.004.

[KLM13] Mamadou Moustapha Kanté, Christian Laforest, and Benjamin Momège. “Trees in Graphs with Conflict Edges or Forbidden Transitions”. In: *Theory and Applications of Models of Computation.* Ed. by T-H. Hubert Chan, Lap Chi Lau, and Luca Trevisan. Springer Berlin Heidelberg, 2013, pp. 343–354. ISBN: 978-3-642-38236-9. DOI: 10.1007/978-3-642-38236-9_31.

[Klu18] Torsten Klug. “Freight Train Routing”. In: *Handbook of Optimization in the Railway Industry.* Springer International Publishing, 2018, pp. 73–91. DOI: 10.1007/978-3-319-72153-8_4.

[Knu98] Donald E. Knuth. *The Art of Computer Programming.* 2nd ed. Vol. Volume 3: Sorting and Searching. USA: Addison Wesley Longman Publishing Co., Inc., 1998. ISBN: 0201896850.

[Kov13] Jakub Kováč. “Complexity of the path avoiding forbidden pairs problem revisited”. In: *Discrete Applied Mathematics* 161.10 (2013), pp. 1506–1512. ISSN: 0166-218X. DOI: `10.1016/j.dam.2012.12.022`.

[KP09] Petr Kolman and Ondřej Pangrác. “On the complexity of paths avoiding forbidden pairs”. In: *Discrete Applied Mathematics* 157.13 (2009), pp. 2871–2876. ISSN: 0166-218X. DOI: `10.1016/j.dam.2009.03.018`.

[KSG73] K. A. Krause, R. W. Smith, and M. A. Goodwin. “Optimal software test planning through automated network analysis”. In: *Proceedings of the IEEE Symposium on Computer Software Reliability.* New York, 1973, pp. 18–22.

[KVB09] Jakub Kováč, Tomáš Vinař, and Broňa Brejová. “Predicting Gene Structures from Multiple RT-PCR Tests”. In: *Algorithms in Bioinformatics.* Ed. by Steven L. Salzberg and Tandy Warnow. Springer Berlin Heidelberg, 2009, pp. 181–193. ISBN: 978-3-642-04241-6. DOI: `10.1007/978-3-642-04241-6_16`.

[LJYZ18] Huiping Liu, Cheqing Jin, Bin Yang, and Aoying Zhou. “Finding Top-k Shortest Paths with Diversity”. In: *IEEE Transactions on Knowledge and Data Engineering* 30.3 (Mar. 2018), pp. 488–502. DOI: `10.1109/TKDE.2017.2773492`.

[MD11] Shi Mu and Maged Dessouky. “Scheduling freight trains traveling on complex networks”. In: *Transportation Research Part B: Methodological* 45.7 (2011), pp. 1103–1123. ISSN: 0191-2615. DOI: `10.1016/j.trb.2011.05.021`.

[Men27] Karl Menger. “Zur allgemeinen Kurventheorie”. ger. In: *Fundamenta Mathematicae* 10.1 (1927), pp. 96–115.

[MG07] Jiri Matousek and Bernd Gärtner. *Understanding and Using Linear Programming.* Springer, Berlin, Heidelberg, 2007. DOI: `10.1007/978-3-540-30717-4`.

[MSWZ07] Matthias Müller-Hannemann, Frank Schulz, Dorothea Wagner, and Christos Zaroliagis. “Timetable Information: Models and Algorithms”. In: *Algorithmic Methods for Railway Optimization.* Ed. by Frank Geraets, Leo Kroon, Anita Schöbel, Dorothea Wagner, and Christos D. Zaroliagis. Springer Berlin Heidelberg, 2007, pp. 67–90. ISBN: 978-3-540-74247-0.

[Opi09] Jens Opitz. *Automatische Erzeugung und Optimierung von Taktfahrplänen in Schienenverkehrsnetzen.* Gabler, 2009. DOI: `10.1007/978-3-8349-8466-1`.

[Pap94] Christos H. Papadimitriou. *Computational complexity.* Addison-Wesley, 1994, pp. I–XV, 1–523. ISBN: 978-0-201-53082-7.

[PY91] Christos H. Papadimitriou and Mihalis Yannakakis. “Optimization, approximation, and complexity classes”. In: *Journal of Computer and System Sciences* 43.3 (1991), pp. 425–440. ISSN: 0022-0000. DOI: `10.1016/0022-0000(91)90023-X`.

[RS95] Neil Robertson and Paul D. Seymour. “Graph minors .XIII. The disjoint paths problem”. English (US). In: *Journal of Combinatorial Theory. Series B* 63.1 (Jan. 1995), pp. 65–110. ISSN: 0095-8956. DOI: `10.1006/jctb.1995.1006`.

[San92] Neville G. F. Sancho. "A dynamic programming solution of a shortest path problem with time constraints on movement and parking". In: *Journal of Mathematical Analysis and Applications* 166.1 (1992), pp. 192–198. ISSN: 0022-247X. DOI: 10.1016/0022-247X(92)90335-B.

[San94] Neville G. F. Sancho. "Shortest Path Problems with Time Windows on Nodes and Arcs". In: *Journal of Mathematical Analysis and Applications* 186.3 (1994), pp. 643–648. ISSN: 0022-247X. DOI: 10.1006/jmaa.1994.1324.

[Suu74] J. W. Suurballe. "Disjoint paths in a network". In: *Networks* 4.2 (1974), pp. 125–145. DOI: 10.1002/net.3230040204.

[Tan14] Yuyan Tan. "Techniques for Inserting Additional Train Paths into Existing Cyclic Timetables". PhD thesis. TU Braunschweig, 2014.

[Vyg95] Jens Vygen. "NP-completeness of some edge-disjoint paths problems". In: *Discrete Applied Mathematics* 61.1 (July 1995), pp. 83–90. DOI: 10.1016/0166-218X(93)E0177-Z.

[Wes00] Douglas B. West. *Introduction to Graph Theory*. 2nd ed. Prentice Hall, Sept. 2000. ISBN: 0130144002.

[WON14] Reyk Weiß, Jens Opitz, and Karl Nachtigall. "A Novel Approach to Strategic Planning of Rail Freight Transport". In: *Operations Research Proceedings 2012*. Ed. by Stefan Helber, Michael Breitner, Daniel Rösch, Cornelia Schön, Johann-Matthias Graf von der Schulenburg, Philipp Sibbertsen, Marc Steinbach, Stefan Weber, and Anja Wolter. Cham: Springer International Publishing, 2014, pp. 463–468. ISBN: 978-3-319-00795-3. DOI: 10.1007/978-3-319-00795-3_69.

[WS88] Ady Wiernik and Micha Sharir. "Planar realizations of nonlinear davenport-schinzel sequences by segments". In: *Discrete & Computational Geometry* 3.1 (Mar. 1988), pp. 15–47. ISSN: 1432-0444. DOI: 10.1007/BF02187894.

[XK02] Zhaoyun Xing and Russell Kao. "Shortest path search using tiles and piecewise linear cost propagation". In: *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 21.2 (Feb. 2002), pp. 145–158. ISSN: 1937-4151. DOI: 10.1109/43.980255.

[Yin97] Hananya Yinnone. "On paths avoiding forbidden pairs of vertices in a graph". In: *Discrete Applied Mathematics* 74.1 (1997), pp. 85–92. ISSN: 0166-218X. DOI: 10.1016/S0166-218X(96)00017-0.

# Curriculum Vitae

## Tim Bergner

| | |
|---|---|
| since Feb. 2019 | **Doctoral Studies in Mathematics**, Technische Universität Kaiserslautern (TUK) |
| December 3, 2018 | **Master of Science in Mathematics**, TUK |
| Apr. 2016 – Dec. 2018 | **Master Studies in Mathematics**, TUK |
| April 12, 2016 | **Bachelor of Science in Mathematics**, TUK |
| Apr. 2013 – Apr. 2016 | **Bachelor Studies in Mathematics**, TUK |
| March 16, 2013 | **Abitur**, Werner-Heisenberg-Gymnasium Bad Dürkheim |

# Wissenschaftlicher Werdegang

## Tim Bergner

| | |
|---|---|
| seit Feb. 2019 | **Promotionsstudium in Mathematik**, Technische Universität Kaiserslautern (TUK) |
| 3. Dezember 2018 | **Master of Science Mathematik**, TUK |
| Apr. 2016 – Dez. 2018 | **Masterstudium in Mathematik**, TUK |
| 12. April 2016 | **Bachelor of Science Mathematik**, TUK |
| Apr. 2013 – Apr. 2016 | **Bachelorstudium in Mathematik**, TUK |
| 16. März 2013 | **Abitur**, Werner-Heisenberg-Gymnasium Bad Dürkheim |