Doctoral Thesis

On the Impact of Uncertainty on some Optimization Problems
Combinatorial Aspects of Delay Management and Robust Online Scheduling

Author(s):
Gatto, Michael J.

Publication Date:
2007

Permanent Link:
https://doi.org/10.3929/ethz-a-005593949

Rights / License:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection. For more information please consult the Terms of use.
On the Impact of Uncertainty on some Optimization Problems:
Combinatorial Aspects of Delay Management and Robust Online Scheduling

A dissertation submitted to the
Swiss Federal Institute Of Technology, ETH Zurich
for the degree of Doctor of Sciences

presented by
Michael Joseph Gatto
Dipl. Inf. Ing. ETH Zurich
born September 12, 1977, citizen of Bönigen, Bern, Switzerland

accepted on the recommendation of
Prof. Dr. Peter Widmayer, ETH Zurich, examiner
Prof. Dr. Anita Schöbel, Georg-August-Universität Göttingen, co-examiner

2007
Abstract

Real-life optimization problems frequently rely on input data which only estimates the data actually appearing when a computed solution is used in practice. Small deviations of the actual data from the input data can have many different consequences: for instance, a computed solution might need to be adapted in order to guarantee feasibility, or the quality of a computed solution might degrade significantly. In some cases, a computed solution can be adapted to incorporate the deviations, thus making the solution feasible again or qualitatively better. In other cases, it might be too late to perform adaptations, and the quality of the solution irredeemably suffers from the deviations. In this thesis, we consider two problems dealing with such perturbations of the input data.

The first problem considered is delay management in railway systems, a problem arising when trains run late. Delay management consists in deciding which trains should wait for delayed transferring passengers, with the goal of minimizing the passenger discomfort. If a connecting train waits for delayed passengers, these passengers are able to board the connecting train, but delay propagates to all the passengers using the connecting train. As a consequence, these passengers might also miss a future connection. If the connecting train departs as scheduled, the delayed transferring passengers miss the connection, and have to board the next available connection, which results in a bigger delay for them. Delay management aims at finding a trade-off between delaying passengers which were originally on time, and letting delayed passengers miss their connections. Currently, this aspect of railway operations is usually handled by human dispatchers. This thesis analyzes a theoretical model for delay management. This model is highly simplified and not directly applicable in practice. Because of its simplicity, the model allows to focus on the propagation of delays through the network. For this focus and the offline case where the complete input is given in advance, we show that variations of the following parameters have a significant impact on the computational complexity of delay management: the maximum number of transfers over all passengers; the structure of the railway network; the possibility of trains to catch up on their delay; the existence of intermediate stops. The proofs of hardness are complemented with polynomial-time algorithms. We also address the online case, where the information about the delays of passengers is
revealed sequentially. We relate restricted versions of online delay management to variants of the ski-rental problem, give competitive online algorithms and show lower bounds on the competitive ratio.

The second problem addresses the degradation of the quality of a solution computed with Graham’s algorithm for the online parallel machine scheduling problem. The perturbations change the processing times and it is not possible to adapt the solution. The case study focuses on an online problem as a setting where decisions are irrevocable. Parallel machine scheduling with Graham’s algorithm is a prime example of a simple and otherwise well understood online problem. The focus is on a worst-case analysis as a probability distribution of the processing times is not assumed to be known. The quality of a solution is measured in a fashion similar to competitive analysis. First, the assignment of jobs to machines is computed using Graham’s algorithm with the unperturbed processing times; then, the processing times are substituted with the perturbed ones. The value of this solution is compared with the value of an optimal solution computed in an offline fashion and knowing the whole perturbed input in advance. The analyses consider the worst-case ratio of these values. The deviation of the ratio from the competitive ratio of Graham’s algorithm shows how much perturbations affect the quality of the solutions. We analyze the impact of either increasing or decreasing the processing times of a limited number of jobs arbitrarily, or by a bounded factor of the unperturbed processing times. For these perturbations, we show upper bounds for the quality loss of the schedules obtained with Graham’s algorithm, and provide a family of examples and perturbations for which Graham’s algorithm matches these bounds or comes close. We also provide simple lower bounds on the best-possible quality achievable by any online algorithm, and propose an algorithm that is tailored for bounded increases of the processing time of one job.
Riassunto

Di frequente, le soluzioni calcolate per i problemi del mondo reale si basano su dati che rappresentano una stima dei dati che si verificano in pratica, quando la soluzione viene poi utilizzata. Le piccole differenze tra i dati reali e quelli utilizzati per il calcolo hanno diverse conseguenze: ad esempio, la soluzione può risultare non più ammissibile se non viene adattata alle perturbazioni, o la qualità della soluzione può risentirne in modo rilevante. In alcuni casi, è possibile adattare le soluzioni calcolate in modo da tener conto delle differenze dell’input; questi adattamenti permettono di rendere una soluzione nuovamente ammissibile, o di migliorarne la qualità. In altri casi la soluzione non può più essere adattata, cosicché la qualità della soluzione risente delle perturbazioni in modo irrimediabile. In questa tesi consideriamo due problemi che si occupano di perturbazioni di dati.

Il primo di questi problemi, quello del delay management per sistemi ferroviari, sorge quando i treni accumulano un certo ritardo. Tale problema consiste nel decidere quali treni debbano aspettare per garantire le coincidenze ai passeggeri a bordo dei treni in ritardo. L’obiettivo è quello di minimizzare il disagio dei passeggeri. Se un treno aspetta, i passeggeri in ritardo riescono a prendere quel treno, ma il ritardo di questi passeggeri viene propagato agli altri passeggeri del treno che aspetta. Di conseguenza, anch’essi rischiano di perdere una coincidenza. Se il treno invece non aspetta, i passeggeri ritardatari perdono la coincidenza e dovranno prendere il treno successivo. Questo porta ad un ritardo ancora maggiore per questi passeggeri. Lo scopo del delay management è quello di trovare il giusto equilibrio tra i passeggeri inizialmente puntuali a cui viene imposto un ritardo, e quelli che perdono una coincidenza. Attualmente, questo aspetto di pianificazione viene svolto da persone. Nella presente tesi, analizziamo un modello semplificato per questo problema. Tale modello non è adatto a risolvere il problema reale, ma ci permette di concentrarci su una caratteristica essenziale del problema stesso: la propagazione dei ritardi. Con questo obiettivo in mente, consideriamo il problema offline dove l’istanza completa è nota a priori, e mostriamo che i parametri che rendono il problema computazionalmente intrattabile sono: il numero massimo di coincidenze previste per ciascun passeggero; la struttura della rete ferroviaria; la capacità dei treni di recuperare parte del ritardo; l’esistenza di fermate inter-
medie. Per alcuni parametri, proponiamo degli algoritmi polinomiali per i casi non $NP$-completi. Ci occupiamo anche del caso online, in cui l’informazione riguardante il ritardo dei passeggeri viene rivelata sequenzialmente. Per questa variante, mettiamo in relazione casi speciali del problema del delay management con alcune generalizzazioni del problema del noleggio sci, proponiamo degli algoritmi con competitività garantita e dimostriamo limitazioni inferiori al rapporto di competitività di ogni algoritmo.

Il secondo problema considerato riguarda la perdita di qualità delle soluzioni calcolate con l’algoritmo di Graham per il problema classico dello scheduling online quando parte dei tempi di esecuzioni sono soggetti a perturbazioni. Consideriamo un problema online quale esempio di un problema dove le decisioni prese sono irrevocabili, e ci concentriamo su un problema di ottimizzazione online ampiamente studiato, ovvero quello dello scheduling su macchine parallele, a cui applichiamo l’algoritmo di Graham. Consideriamo un’analisi di tipo worst-case visto che assumiamo che non sia nota una distribuzione probabilistica dei tempi di esecuzione. La qualità di una soluzione viene misurata in una maniera del tutto analoga a quanto viene generalmente fatto per l’analisi di competitività. Inizialmente, i lavori vengono assegnati alle macchine usando l’algoritmo di Graham basato sui tempi di esecuzioni non perturbati. In questa soluzione, sostituiamo i tempi di esecuzione con quelli perturbati, e confrontiamo questa soluzione con la soluzione di un algoritmo ottimo offline che conosce i tempi di esecuzioni perturbati a priori, e ci concentriamo su un’analisi di tipo worst-case. La deviazione del rapporto di queste soluzioni dal rapporto di competitività garantito dall’algoritmo di Graham ci dice quanto la qualità della soluzione risenta delle perturbazioni. Tale deviazione viene analizzata nei due scenari seguenti: quando incrementiamo o decrementiamo in modo arbitrario il tempo di esecuzione di un numero ristretto di lavori, e quando perturbiamo i tempi di esecuzione di un dato fattore. Per questi casi, dimostriamo delle limitazioni superiori alla perdita di qualità dell’algoritmo di Graham, e forniamo famiglie di esempi e di perturbazioni per le quali l’algoritmo ottiene tali perdite di qualità, o vi arriva vicino. Dimostriamo anche semplici limitazioni inferiori alla perdita di qualità minima ottenibile con un qualsivoglia algoritmo online, e proponiamo un algoritmo online per il caso in cui il tempo di esecuzione di un lavoro incrementa al più di un dato fattore del tempo di esecuzione dichiarato.
Acknowledgments

Seven weeks after my defense, I still have to write the acknowledgments. Thanksgiving in the United States of America seems like a good day to do so.

First of all, I’d like to thank my supervisor Peter Widmayer for making this thesis possible, for the interesting discussions we had, for his support, and for his perseverance at wanting to understand delay management in spite of me loosing faith and feeling like doing other things. Also, he enabled me to conclude this thesis quickly, in time for an interesting PostDoc position. Furthermore, I’m grateful for him jumping into my office with the scheduling problem, which turned out to be great fun and worth a trip to Canada. Finally, I’d like to thank him for the great atmosphere at the office.

I am also very grateful to Anita Schöbel for agreeing to be my co-examiner on such a short notice, for doing the required reading and paperwork within ten days, and for her never-ending enthusiasm for delay management. Also, I enjoyed the interesting questions during the defense and the work we did together.

A big thank you goes to every colleague and former colleague of the group, and not only for the work we did together. They made a very pleasant atmosphere in the group, which made the working environment great. Specifically, and in no particular order, I’d like to thank: Gabor, for being a great office mate in CLW and for sharing his interest in photography with me; Marc, for the cool work we did together and the four o’clock in the morning-session, and the countless parties he organized in Zurich and Berlin; Luzi, for the cool cappuccino sessions, and the discussions about sport and free-time stuff; Franz, for our interesting discussions; Conrad, for believing in his work; Mirjam, for trashing me at Singstar; Barbara, for Gipfeli, discussions and helping us when we needed it; Fabian, for introducing me to Beresina; Birgitta, for the good cakes; Riko, for working with me and for his crazy setup to photograph water drops; Leon, for his constant advice and support, for reading my stuff and returning it red with ink, for the whiskey tasting nights, for sharing his biking knowledge and his enthusiasm about everything he does; Matus, for his positive thinking, for having a smile and a kind word for everybody, and for proof-reading part of this thesis. Elias, for being “Joga” and for the linux-and-geek-talk in Italian; Holger, for intro-
ducing the “Monatsbier” and making me drive a SMART; Michele, for the coffee breaks and the ETH-anecdotes; Beni, also for the coffee breaks and for tolerating my comments about architects; Jörg, for telling me that the D+A-Chefassistenz would not be so much work; Beat, for taking over the D+A-Chefassistenz from me (really! It’s not so much work!); Alex, for his skills at table soccer; Jens, for his sense of humor; Tomas, for the technical discussions during coffee breaks; Marianna, for the free coffee; Davide, for showing me what a real Italian is; Anna, for rising the highly needed women-quota of the group; Rasto, for taking over the geek position.

During my PhD, I also had the pleasure to work with Guido, Anita, Michael, Alberto, Heiko, Marco and Mr. Potato Luciano. A big thank you for the research and for the pleasant time we spent together. In particular, I thank Mr. Potato for reading through my Italian abstract and for preventing me from making too much of a fool of myself by rephrasing it. Also, I thank my father for proof-reading the introduction of this thesis.

My time at ETH would not have been so pleasant without the many friends outside the group. Thanks to the too-many-to-mention girls and guys somehow related to Pranzo, Pettegolezzi, Rpg, Sport and Cinema, the swimming X-Chromosome, and the friends near and far. You allowed for a balanced environment when I was not working!

A big thank you also goes to my flatmate Paolo, who supported me by cooking countless evening meals when I was too busy writing up or doing something else, and by chatting about stuff more important than work whenever possible.

I’m grateful to my sister Dominique for keeping my spirits up during my thesis, having gone through one before me, and for the big help she and Sascha gave me in the hectic times around my defense, and for coming to my defense in the first place in spite of a 24-hour flight.

Infine, un grande ringraziamento va a te, Micky, per aver sopportato pazientemente tutti quegli umori-da-venerdì-sera e i sono-altrove-con-la-testa; per il tuo costante supporto, morale e non; e per avermi distolto dal dottorato durante i week-end. Non da ultimo, grazie per aver sempre vissuto con me il mio dottorato e per il tuo sostegno incondizionato in tutte le decisioni che prendo. Grazie!
## Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Abstract</td>
<td>iii</td>
</tr>
<tr>
<td>Riassunto</td>
<td>vii</td>
</tr>
<tr>
<td>Acknowledgements</td>
<td>ix</td>
</tr>
<tr>
<td>Contents</td>
<td>xi</td>
</tr>
</tbody>
</table>

### 1 Introduction

1.1 Thesis outline and summary of results                               5

1.2 Short tour through algorithm and complexity theory                  6

### 2 A model for delay management

2.1 Related work                                                        13

2.2 A general model                                                     17

2.3 Models within this thesis                                           24

2.3.1 Restrictions common to all considered models                      24

2.3.2 Problem statements                                                29

2.3.3 Delay management with slack times                                 31

2.4 Restrictions, equivalences and a first reduction                    32

2.4.1 Intermediate stops                                               34

2.4.2 Delayed passenger paths                                           37

2.4.3 Dynamic path choices                                              39
3 Offline delay management

3.1 Hardness of the binary delay management problem
   3.1.1 Proof of hardness with three transfers
   3.1.2 Single delayed leg
   3.1.3 Dynamic passenger path choices
   3.1.4 Approximating the additional passenger delay

3.2 Minimum-cut based approaches
   3.2.1 Passenger paths with at most two transfers
   3.2.2 Special cases
   3.2.3 Fixed parameter tractability

3.3 Dynamic programming for a railway corridor
   3.3.1 Enumerating all delay policies
   3.3.2 Pruning equivalent subtrees
   3.3.3 A dynamic programming view
   3.3.4 Extensions of the algorithm

3.4 Train services with intermediate stops
   3.4.1 Proof of hardness
   3.4.2 Polynomial-time solvable cases

3.5 Binary delay management problem with slack times
   3.5.1 Proof of hardness
   3.5.2 Polynomially solvable cases

4 Renting skis and managing delays online

4.1 Ski rental and online algorithms
   4.1.1 Discounted ski rental problem
   4.1.2 The generalized $k$-day ski rental problem

4.2 Online delay management
   4.2.1 Model, notation and online information disclosure
   4.2.2 Delay management and ski rental

4.3 Competitive online algorithms
   4.3.1 A family of 2-competitive algorithms for a single train on a corridor
## Contents

4.3.2 From a single train to multiple trains on a corridor .......................... 116
4.3.3 From a corridor to tree-like networks with a common terminal leg ........... 120
4.4 Lower bounds on the competitive ratio .............................................. 122
4.4.1 Golden ratio competitiveness ......................................................... 122
4.4.2 Ski-rental-like lower bounds ......................................................... 125
4.5 Extensions and Limitations ............................................................... 127
4.5.1 Randomization using $A(s)$ ............................................................ 127
4.5.2 Lower bounds for the additional passenger delay objective ................. 128
4.5.3 Limits of ski-rental-like approaches ............................................... 130

5 Robust Online Scheduling 135
5.1 Problem setting and notation ............................................................. 141
5.2 Arbitrary perturbations ................................................................. 143
  5.2.1 Arbitrary decreases in processing time ....................................... 143
  5.2.2 Arbitrary increases in processing time ....................................... 148
5.3 Bounded perturbations ................................................................. 154
  5.3.1 Bounded decreases in processing times .................................... 154
  5.3.2 Bounded increases in processing times .................................... 159
5.4 A Graham-like risk aversion algorithm ........................................... 169

6 Conclusion 177

Glossary 179

Bibliography 189
Chapter 1

Introduction

To many Ticinesi students living in Zurich, outside the borders of their beloved canton, trains become a second home. Not only do the railways help us bring the dirty laundry to mum’s to get it washed in time for the week to follow but, in our case, they also bridge the alps, bringing us to old friends, good food and better coffee than you can generally get in Zurich. Although the need to go back to one’s roots decreases over the years, at least in my case this is so, regular train trips towards the southern part of Switzerland remain a must. Alas, as Swiss railway commuters know very well, not even the Swiss railways are perfect. It is indeed not uncommon for the Cisalpino, the fast train connecting Italy and Switzerland, to be delayed on its way south, thus jeopardizing the connection for the regional train in Arth-Goldau. To the great annoyance of the passengers wishing to connect to the regional train, depending on the day and on the extent of the delay, such connection is not always guaranteed. But even when the connecting train waits, it’s not without discomfort to other passengers. The passengers in the regional train now depart with a delay, since the train has to wait in order to guarantee the connection and, as a result of the delay in departure, some connections of the regional train (such as regional buses) are now in question as well. Thus, the question arises, when should a connecting train wait or not wait for delayed passengers?

The problem sketched above, delay management, is a main stone of this thesis. Delay management addresses the question of which
trains should wait to guarantee a connection to a delayed feeder train, and which ones should leave as scheduled, thus resulting in some passengers missing their connection. This problem can be addressed with many different focuses. In general the main motivation for a railway operator to maintain a connection is to keep the quality of service from degrading too much in spite of some delays that inevitably arise within its operations. However, the railway operator may not be able to guarantee every connection due to limitations in infrastructure; making a train wait for delayed passengers might not always be feasible. For example, the platform where the train is waiting might not be available for a longer than the scheduled period of time. These and other operational constraints result in a variety of different aspects of delay management which can be addressed. In this thesis, we address delay management only from a customer oriented perspective and disregard operational aspects that might constrain our choices on which connections to guarantee. Our goal is to choose the connections to be maintained in such a way that the overall discomfort faced by the passenger is minimized. We quantify this discomfort, taking into account all passengers concerned, by summing the delay with which each passenger arrives at his/her final destination with respect to the time he/she was originally scheduled to arrive.

Our lack of consideration of operational constraints can be justified in two ways. First of all, if the operator has large operational capacities, such constraints might result in being irrelevant. Secondly, disregarding them allows us to focus on the decision of which connections should be maintained or not. Given that we gain insight on the structure of the connections that should be maintained when trains are delayed, a railway operator could then adapt his operations in such a way as to make decisions to wait feasible most of the time. Finally, we restrict our view on the impact of delay management on customers so that we can concentrate on one single aspect of the complex problem. In fact, despite the fact that delay management has been tackled with many solution approaches in other studies, only little insight about its combinatorial structure has so far been gained.

Delay management is a reactive problem. Indeed, the question of which connections to maintain only arises if, due to some unforeseeable event, some trains travel later than expected, thus compromising the connections to other trains. Hence, it is not a part of the planning process as, for instance, determining the railway timetable is; it arises
due to the everyday operation of the railway system and is caused by unexpected perturbations. However, decisions taken in the planning process do indeed have an impact on which connections are endangered in case of delays. The optimization within the planning process in such a way that the plan behaves “well” for small perturbations is known as robust optimization, and is beyond the scope of this thesis for delay management.

In this thesis, we analyze the aspect of robustness for a problem which does not directly come from railway optimization. Our goal is to understand the impact of small perturbations of the input (loosely speaking, of the data we are given to do the planning) when the extent of the unexpected perturbations is known only after the planning process has finished, and we cannot react to the perturbations by adapting the plan (which is, on the contrary, what delay management is all about). We give a small explanatory example hereafter. We point out that we are analyzing the impact of perturbations on the quality of a given solution, where the solution in hand was computed using the unperturbed data.

As has been indicated the issue in hand is the inability to react to perturbations. There are many practical problems where such a situation arises; for example, consider the following problem. An international express courier company has a fleet of trucks which pick the parcels up from the customers and bring them to the airport, where they are flown to another country for delivery. As we are talking about an express service, the company has interest in ensuring that the airplane is able to depart punctually in its departure slot. Thus, each morning it is interested in a set of routes, one for each truck, such that the time when the last truck arrives at the airport is the earliest possible. Each route starts at the company’s headquarters and ends at the airport. Furthermore, a route specifies the parcels a specific truck should pick up, and the sequence to be followed for pickup. Naturally, the time it takes for a route depends on the locations of the parcels, since a route involves traveling along roads. All trucks leave the company’s headquarters at the same time, and each follows its route to collect the parcels. Naturally, the routes are optimized in such a way that the maximum travel time considering all trucks is the least possible. However, these travel times are only estimates: it can easily happen that a truck gets jammed in traffic, and will thus take longer than planned to collect the parcels and ultimately to reach the
airport. It is reasonable to assume that once deployed, the trucks cannot react to these jams, in the sense that one truck’s task cannot be fulfilled by another member of the fleet, since this would require re-building and altering routes while the drivers are already on the road. For this problem, the impact of the perturbations can be measured by comparing the arrival time at the airport of the last truck given the perturbations, with the arrival time of the last truck of an optimal solution that knows the actual (perturbed) traveling times (i.e., it knows about the traffic jams in advance).

In a case such as the above our interest is twofold: on the one hand, we want to measure the impact of perturbations on the quality of the solution at hand, or better still, to derive bounds on the worst case impact of the perturbations on the quality of the solutions obtained with a specific algorithm. On the other hand, we would like to design algorithms which are robust against specific types of perturbations, where by robust we mean that the quality of the solutions computed by the algorithms does not degrade too much when the specific types of perturbations arise.

The field of online optimization restricts the decisions to be irrevocable. In online algorithms, the instance the algorithm needs to optimize is not known, as a whole, from the start. An online algorithm is given chunks of the instance one after the other. Each time the algorithm is given a new portion of the instance, it must take a decision regarding it. Stock market is an online problem: when facing the decision of buying a share, the broker does not know how the prices of that stock will evolve, and must decide, on behalf of the seen prices, when he should buy the share.

In an effort to understand the impact of perturbations within this setting, we consider the simple setting of online parallel machine scheduling. To that aim, we analyze the impact of perturbing the processing times of the jobs in several different ways, and bound the worst-case effect of the perturbations for schedules obtained with Graham’s algorithm with respect to the optimum offline solution of the perturbed instance. We also provide simple bounds on the quality of the solution of any online algorithm for some types of perturbations. For one specific type of perturbation, we also propose a Graham-like online algorithm tailored for this kind of perturbation and analyze it.
1.1 Thesis outline and summary of results

The outline of this thesis is as follows:

In the remainder of this chapter we summarize the results of this thesis, and give a short overview of the most important notions of complexity theory that are required in this thesis.

In Chapter 2 we discuss the related work on delay management and describe the model for delay management that we address. We first introduce a general model, based on the event-activity representation of railway networks. Then, we explain and motivate the simplifying restrictions we introduce into the model. These restrictions allow us to focus exclusively on the aspect of which connections to guarantee. Furthermore, we discuss some modeling issues in more detail and give a first hardness result for a particular delay management setting.

Chapter 3 addresses the offline variant of delay management. We show that several decision versions of the delay management problem are $\mathcal{NP}$-complete even with very severe restrictions on the railway network topology and on the way passengers travel. We also give two combinatorial approaches that solve, in polynomial time, restricted versions of the delay management problem. One version restricts the topology of the railway network to a railway corridor; the other constrains the number of times passengers may transfer. The polynomial time algorithms complement our hardness results. We extend the hardness results for more complex settings of delay management, which result in even more severe restrictions on railway topology or number of passenger transfers. The extensions address the case where trains can catch up on their delay, or where trains have a slightly more complex way of operation. Again, we complement these results with polynomial time approaches for solving restricted problem settings.

We address the online version of delay management in Chapter 4. There, we restrict our view to the artificial and simple setting where the railway network is a corridor. For this setting, we derive lower bounds on the competitive ratio for any online algorithm and propose ski-rental-like online algorithms which achieve the ratio of 2. We then extend these algorithms to manage more complex network structures, while enforcing a fixed common destination for every passenger. As ingredients in the competitive analysis of our online delay management problems, we analyze different variants of the ski rental
problem, and relate them to our delay management problems. We also point out the limitations of the ski rental like approaches.

Chapter 5 is devoted to the analysis of the robustness issues of online parallel machine scheduling. First, we give a revision of the existing robustness results for the general context of scheduling. Then, we introduce the problem variant we address and bound for different types of perturbations, the worst-case effects on the competitive ratio for makespan minimization when using Graham’s algorithm. We show that many of the given bounds are tight. We also give lower bounds on the best-possible competitive ratio of any algorithm dealing with specific perturbations. For instance, we show that by arbitrarily decreasing the processing time of one job, Graham’s algorithm achieves a competitive ratio of 2, and that this is best-possible. We give similar results for the cases of perturbing the processing time of a restricted number of jobs by a constant factor. For instance, we show that by doubling the processing times of two jobs the competitive ratio of Graham’s algorithm is slightly less than 3. Finally, we propose a tailor-made Graham-like online algorithm for the perturbation which doubles the processing time of at most one job. We show that this algorithm does not achieve a better result than the standard online algorithm by Graham.

The results within this thesis are joint work with many co-authors. The precise contributions of my fellow researchers are specified at the beginning of each chapter.

1.2 Short tour through algorithm and complexity theory

In this section, we give a brief overview of the main notions of complexity theory used in this thesis. A thorough review of these notions is beyond the scope of this thesis, since there are many good books devoted solely to this topic. We refer to the following books [6, 20, 46, 59, 65] for an in-depth introduction to the many facets of complexity theory. We also omit an introduction to the notions of basic graph theory, and define specific problems and concepts when needed throughout the thesis. We refer to [16, 35] for in-depth introductions to graph theory.

For the following definitions, we follow the exposition in [6].
1.2. Short tour through algorithm and complexity theory

Definition 1.1 (Problem). A problem $P$ is a relation between the set of problem instances $I$ and the set $S$ of problem solutions.

The size $|I|$ of an instance $I \in I$ is the number of bits necessary to represent the instance. The running time of an algorithm is the number of unitary operations the algorithm performs to evaluate the received input. In this thesis, we assume a Random Access Machine model of computation as discussed in [16]. Thus, the statements about running times and time complexities are with respect to this model of computation. We use the classical $O$-notation to quantify running times (see, e.g., [16]).

Many problems considered in complexity theory have a binary character, in the sense that a problem instance either has some characteristic, or it does not.

Definition 1.2 (Decision problem). A problem $P_d$ is a decision problem if its set of instances $\mathcal{I}_d$ is partitioned into two sets, the set of “yes”-instances $\mathcal{I}_d^Y$ which satisfy the relation and the set of “no”-instances $\mathcal{I}_d^N$ which do not satisfy the relation. Thus, the set of problem solutions is $S = \{\text{Yes}, \text{No}\}$, and the decision problem asks to verify, for $I \in \mathcal{I}_d$, if $I$ is an instance that satisfies the relation, i.e., asks if $I \in \mathcal{I}_d^Y$.

The following two definitions aim at classifying the problems in terms of their computational complexity.

Definition 1.3 (The class $\mathcal{P}$). A decision problem $P_d$ is in the complexity class $\mathcal{P}$ if there is a deterministic polynomial time algorithm $A$ which determines, for all instances $I$ of $P_d$, if $I \in \mathcal{I}_d^Y$.

Intuitively speaking, a problem is in the complexity class $\mathcal{P}$ if the solution of every instance of the problem can be computed deterministically in polynomial time.

A certificate $c$ of a problem instance $I \in \mathcal{I}_d^Y$ is a witness of the fact that $I$ is a “yes”-instance. A verifier $V$ is an algorithm which given an instance $I$ of problem $P$ and a certificate $c$ returns one of two values \{yes, no\} and has the following property: if $I$ is a “yes” instance, there the verifier $V$ returns “yes” for any certificate $c$ for $I$; if $I$ is a “no” instance, then $V$ returns “no” for all possible inputs of $c$. Intuitively speaking, the verifier is able to check that the certificate $c$ is a witness of the fact that $I$ is a “yes”-instance.
Definition 1.4 (The class \(\mathcal{NP}\)). A problem \(P\) is in the complexity class \(\mathcal{NP}\) if there exists one verifier \(V\) for all pairs of an instance \(I\) of \(P\) and of a polynomial size certificate \(c\) for \(I\) which runs in polynomial time.

Intuitively, the class \(\mathcal{NP}\) contains all problems whose solutions can be checked in polynomial time.

Definition 1.5 (Polynomial time reduction). A decision problem \(P\) reduces to another decision problem \(P'\), written \(P \leq P'\), if and only if there is a polynomial time algorithm that maps any instance \(I\) of \(P\) to an instance \(I'\) of \(P'\) in such a way that \(I\) is a “yes”-instance if and only if \(I'\) is a “yes”-instance.

Intuitively, this means that the problem \(P'\) is at least as hard as \(P\). Often, we say that we reduce from a problem \(P\), meaning that we give an algorithm \(P \leq P'\) which transforms any instance of a problem \(P\) into a corresponding instance of the problem \(P'\).

Definition 1.6 (\(\mathcal{NP}\)-hard). A decision problem \(P'\) is said to be \(\mathcal{NP}\)-hard if every decision problem \(P \in \mathcal{NP}\) can be reduced to \(P'\).

Definition 1.7 (\(\mathcal{NP}\)-complete). A decision problem \(P'\) is said to be \(\mathcal{NP}\)-complete if \(P'\) is in the class \(\mathcal{NP}\), and every other decision problem \(P \in \mathcal{NP}\) can be reduced to \(P'\), i.e., \(P \leq P'\).

Intuitively, this means that \(\mathcal{NP}\)-complete problems are the hardest in the class \(\mathcal{NP}\). The widely believed assumption that \(\mathcal{NP} \neq \mathcal{P}\) excludes the existence of polynomial time algorithms for \(\mathcal{NP}\)-complete problems.

Decision problems play an important role in the classification of problems according to how difficult they are to solve. However, in practice we often search for a solution to a problem which satisfies some relation, and each solution has some form of cost which we want to minimize or maximize. Thus, we have the following definition [6].

Definition 1.8 (Optimization problem). An optimization problem \(P\) is a quadruple of objects \((\mathcal{I}_P, \text{SOL}_P, m_P, \text{goal}_P)\) defined as follows:

- \(\mathcal{I}_P\) is the set of instances of \(P\);
1.2. Short tour through algorithm and complexity theory

- \( \text{SOL}_P \) is a function associating the set of feasible solutions to any input instance \( I \in \mathcal{I}_P \);

- \( m_P \) is a function that for every pair \( (I, S_I), I \in \mathcal{I}_P, S_I \in \text{SOL}_P(I) \), quantitatively measures the quality of the solution \( S_I \) for the input \( I \).

- \( \text{goal}_P \in \{\text{min}, \text{max}\} \), specifies if the qualitative measure of the solution is to be maximized or minimized in for the problem \( P \).

Thus, in an optimization problem we aim at minimizing or maximizing a certain measure defined for the solutions. The function \( m_P \) is often referred to as the objective function. The solution which attains the minimum (for minimization problems) or the maximum (for maximization problems) objective value is called the optimum solution. The notion of \( \mathcal{NP} \)-hardness also exists for optimization problems, and can be defined with respect to associated decision problems. In this definition, taking a minimization problem as an example, the question being addressed is of the following type: does a solution of the optimization problem exist, such that the value of the measure function \( m_P \) is not greater than a value \( k \) given in the input? A formal definition is given for instance in \([6]\). The \( \mathcal{NP} \)-completeness of the decision problem associated with an optimization problem also excludes the existence of polynomial time algorithms for the optimization problem (under the assumption \( \mathcal{P} \neq \mathcal{NP} \)). In an effort to find solutions with a quality guarantee, research has widely addressed polynomial time approximation algorithms, defined in the following.

**Definition 1.9 (Approximation algorithm).** Let \( P \) be an optimization problem with a strictly positive objective function, and let \( A \) be a polynomial time algorithm delivering, for every instance \( I \in \mathcal{I}_P \), a feasible solution \( s_A(I) \in \text{SOL}_P(I) \). The algorithm \( A \) is said to be an approximation algorithm with approximation ratio \( r \) if, for every instance \( I \in \mathcal{I}_P \):

\[
\max_{I \in \mathcal{I}_P, s' \in \text{SOL}_P(I)} \left\{ \frac{m_P(I, s')}{{m_P(I, s_A(I))}}, \frac{m_P(I, s_A(I))}{{m_P(I, s')}} \right\} \leq r.
\]

The approximation ratio \( r \) can be of many different forms: a constant in \( \mathbb{Q}_0^+ \), a function in the size of the input, or the two combined. Some problem do not admit any approximation algorithm. We call a problem inapproximable if there exists no algorithm \( A \) that, for any
Chapter 1. Introduction

given instance, has a bounded approximation ratio. Results of this type arise when addressing approximation algorithms for problems which may have a zero-valued objective function (thus the specification about the objective function in the above definition). In this thesis, we briefly address an approximation algorithm of the following form:

**Definition 1.10 (FPTAS).** An approximation algorithm $A$ for an optimization problem $P$ is a Fully Polynomial Time Approximation Scheme (FPTAS) if for every given pair built by an instance $I \in \mathcal{I}_P$ and by $\varepsilon \in \mathbb{Q}_0^+$, it achieves an approximation ratio of $(1 + \varepsilon)$ in time polynomial in $|I|$ and $\frac{1}{\varepsilon}$.

**Definition 1.11 (Fixed parameter tractable).** Let $g$ be an arbitrary function. An algorithm for an optimization problem is called fixed parameter tractable with respect to a parameter $t$ of the problem instance of size $n$ if its running time is bounded by $O(g(t) \cdot n^c)$, for some constant $c \in \mathbb{R}^+$.

In the context of fixed parameter tractable algorithms, the function $g(t)$ is in general some exponential function in $t$. Thus, fixed parameter tractable algorithms are still exponential algorithms, but the exponential growth is scaled down to the parameter $t$ only, and is, up to that exponential growth, polynomial in the size of the rest of the instance. The interested reader is referred to [17] for an introduction on fixed parameter tractability.

Chapters 4 and 5 address online problems. Differently from offline problems described above, in online problems the instance is not disclosed fully from the beginning. Instead, it is presented a portion at a time, and upon receiving each new piece of information, the online algorithm must react to this information before receiving the next portion of the instance. We introduce the necessary theory on online algorithms and on competitive analysis in Section 4.1 in the context of the ski rental problem. A good guide to the state of the art of online algorithms are the books [18] and [12].
Chapter 2

A model for delay management

It’s so easy but I can’t do it
So risky but I gotta chance it
It’s so funny there’s nothing to laugh about [⋯]
I can see what you want me to be
But I’m no fool
It’s in the lap of the Gods
In the lap of the Gods...revisited, Queen.

Even a carefully planned railway system will once in a while have to deal with delayed trains due to unforeseeable events. In such a situation, some of the train’s passengers may miss a connecting train, resulting in an even larger delay because they have to wait for the next train. A railway operator seeking for high customer satisfaction can react by maintaining some connections and modifying the train schedule accordingly.

The delay management problem considers a trade-off on which connections to maintain that is best explained by an example. Consider a passenger in an on time train that decides to wait for a delayed feeder train. Although the passenger was traveling on time, she now faces a delay because of this decision. Moreover, she herself may later miss a connecting train in a subsequent station. Alternatively, had the train not waited, then the connecting passengers in the feeder train would have missed their connection. In particular, they would have had to wait for the next train, thus facing a large delay each.
In general, it might thus be beneficial to propagate the delays in the network, and one should decide on a set of waiting trains.

Delay management consists of deciding which connecting trains should wait for what delayed feeder trains, usually with the objective of minimizing the overall discomfort faced by the passengers. Even today, decisions to make some trains wait are usually taken by a human dispatcher. Ideally, a modern railway decision support system should enable a dispatcher to easily evaluate the impact of his or her decisions, or even propose a good waiting policy. This dream naturally leads to the algorithmic problem of determining an optimal waiting policy that minimizes the overall passenger delay. Various approaches addressing the rescheduling of disturbed railway operations have been considered in the last years, and are reviewed in Section 2.1.

In the following chapters, we consider delay management with a very biased look. Namely, we only focus on the point of view of the customers, and disregard all operational constraints which enter the game upon a decision to explicitly make one train wait for connecting passengers. These disregarded operational constraints include, for example, platform and track availability, crew and rolling stock rostering, and are shortly discussed in Section 2.4. Thus, our only objective is to optimize the passenger’s connections in such a way that it is overall beneficial for them, and ignore all other issues. Furthermore, we only allow a reactive approach: we assume a timetable to be given, and can only decide which trains should wait for delayed connecting feeder trains. We do not assume a feedback loop that, in case of repeated delays on the same train, would allow to modify the timetable. Moreover, our focus is on small delays, such that all connections could be maintained by slightly delaying each and every train. We do not consider huge delays which can arise from track interruptions or engine breakdowns, as such delays fall in the domain of disruption management (see, for instance, [36] and the literature cited therein).

Outline In this chapter, we first give an overview of the related work within delay management (Section 2.1). Then, we describe the delay management problem formally (Section 2.2 and 2.3) and discuss some of the limitations of the problem setting and motivate some of the choices (Section 2.4).
2.1. Related work

Chapter 3 addresses the complexity question of delay management in the offline setting. In the offline setting of delay management, we assume all delays are given with the instance (as could happen by taking a snapshot of a delay situation), and we seek for an optimal delay policy. In a different perspective, Chapter 4 addresses an online version of delay management. In the online setting, the instance is specified as for the offline setting, with the exception of the source delays of the passengers, which represent the online component. As such, they are given one by one during the execution of an online algorithm.

2.1 Related work

The research addressing railway delays started as early as two decades ago (see, for example, [30]). In contrast to other railway problems which have been thoroughly analyzed (see, for an overview, [5, 13]), many aspects in delay management remain, at the present stage, unclear. In this section, we give a short overview of the many different approaches which have been used to tackle delay management.

The probably most common technique for addressing and modeling delay management is mathematical programming. Many different formulations appeared over the years, some for highly customized settings and used mainly to mathematically model the problem at hand precisely, and some for general settings with the goal, not always practicable, of solving instances to optimality. The objective of transporting the maximum number of passengers with the option of introducing trains to carry out the service of delayed trains and to suppress some trains was analyzed in [1] for the Spanish national railways Renfe. However, this formulation is too complex to be solved in practice, and a backtracking-based heuristic was implemented instead to explore the solution space. In a direction closer to our objective of delay management, [62] analyzed a general model for delay management with the goal of minimizing the overall passenger delay. This integer linear programming formulation assumes a periodic timetable and penalizes waiting times in (warm and cozy) vehicles less than waiting times on (cold and windy) platforms. Furthermore, this model assumes a constant number of passengers transferring between trains (that is, ignoring the fact that some passengers may not have reached some transfer station because they missed a connection...
earlier). Potentially, this assumption may lead to a wrong objective value if passengers transfer more than once. As a solution to the over-counting of passenger delays, the mixed integer linear programming formulation in [54] proposes a path-based approach for passengers: each specific route taken by passengers is considered separately, thus not considering passenger totals for the trains. This model also assumes a periodic timetable and that operations resume to normal in the following timetable cycle, thus causing a fixed delay per passenger in case of a missed connection. This formulation can be solved, for small problem instances, with general-purpose ILP solvers. The results for path-based passenger models were extended in [55], where a general variant of delay management minimizing the total passenger delay is addressed. Several different formulations are proposed (some of which quadratic) and analyzed. The analysis of one of the ILPs shows that if the connections which must be maintained are fixed a priori, the constraint matrix which adapts the timetable with the occurring delays is totally unimodular (for a definition, see for example [56]), and thus the delay management problem is solvable in polynomial time. However, the main question of which connections should be maintained remains unaddressed by this solution approach. In another general formulation of [55], the number of passengers per leg is fixed to the sum of all passengers who intend using that leg. This formulation, which overestimates the number of passengers per leg as some passengers might miss a connection, was shown to produce the correct results to a path-based passenger formulation in a special case and in polynomial time. If in an optimal solution no two delayed vehicles (or vehicle and passenger) meet at any node, the never-meet-property of [55] holds and the formulation with fixed weights provides the correct result. This property immediately holds on instances with a network topology of an out tree and a single initially delayed vehicle. For cases where the never-meet-property holds and trains cannot catch up on their delay, the constraint matrix was shown to be totally unimodular, thus the ILP solvable in polynomial time. Furthermore, the case where trains can catch up on some of their delay was shown to be solvable in polynomial time with a combinatorial approach.

The second prominent approach to tackle delay management has been through heuristics and simulations. A wide spectrum of problems was tackled in this field, most capturing many specific aspects of a particular setting. The simulations vary from implementing the
2.1. Related work

delay policy used on a specific railway network up to a very fine level of detail [10] to just evaluating the extent of the delay for each train [31]. Since our focus is on optimization and computational complexity, we discuss these approaches only very briefly. In the series of papers [62, 64, 63, 10], the authors address the question of which connections should be maintained by applying deterministic decision policies. For example, one of the policies stems from the decision policy used by Deutsche Bahn, which makes a hierarchically less important train wait a predefined amount of time for the more important trains. Another policy tested compares the number of connecting passengers with the number of passengers that would be immediately delayed by the waiting decision. The policies are tested in an agent-based simulation tool on real-world data of Deutsche Bahn. The simulation introduces delays on the trains randomly over time, with an exponential distribution. The results of the simulation allow to draw qualitative conclusions on the different waiting policies. For instance, the simulations show that both maintaining all connections and not maintaining any connection cause a larger total passenger delay than the decision policy of Deutsche Bahn. The latter is outperformed by using more complex decision policies which take passenger information into account.

A different approach for guaranteeing many connections in the case of delays is to include a buffer of additional waiting time at transfer stations directly in the original timetable. Obviously, big buffer times make missing a connection less likely; on the other hand, bigger buffers cause longer traveling times. The series of papers [25, 24] analytically determined the optimal buffer time for the following setting: the arriving train has an exponentially distributed arrival delay and the goal is to minimize the expected waiting time at a transfer station. Hence, this setting basically considers the optimal buffer time for one transfer. An attempt to extend this approach and analytically compute the optimal buffer time for a sequence of more than one transfer showed to be impracticable, and the problem was tackled with a Monte-Carlo simulation [9].

In spite of the lack of efficient algorithms for solving the delay management problem, surprisingly no strong \textit{NP}-completeness results were known for it. The only results in this direction address a bi-criteria variant of delay management. The two criteria to be optimized are the sum of the arrival delays of the vehicles at each sta-
tion and the number of missed passenger connections, both of which should be minimized. Obviously, a good solution for the first criterion is bad for the second one (and vice-versa): The arrival delay of each train is minimized when no vehicle waits for connecting passengers. The two criteria reflect the view of an operator who would like to resume to the planned schedule as soon as possible whilst still providing a decent service for the passengers. This variant of delay management is shown to be weakly \( \mathcal{NP} \)-complete in [55, 23] by reduction from the knapsack problem. For the same bi-criteria problem, [41] provides a slightly different complexity proof and some further theoretical observations.

Naturally, delay management is intrinsically an online problem, since delays appear over time and decisions must be taken immediately and irrevocably without knowledge of further delays. To the best of our knowledge, the only online analysis of a delay management problem is by [4]. This setting considers a bus station with buses arriving at regular time intervals, and passengers arriving with a fixed arrival rate. For this problem, the objective is to decide which buses should wait for how long at the bus station such that the overall passenger waiting time is minimized. The authors derive tight bounds on the competitive ratio of online algorithms for several cases of this problem setting.

Clearly, delay management problems are not exclusive to railways, but rather arise in every public transportation network. However, the view of the planners in each transportation network is different. Public transportation systems with high frequency, such as trams or buses in cities, are less interested in guaranteeing connections, since it is not worth disrupting the schedule for allowing some passengers to wait a couple of minutes less. In general, only very few connections are actively maintained, such as the last connection of the day. The optimization of the number of missed connections is also of concern for airlines. However, here these problems are only of secondary importance, since other problems occurring in case of delayed aircrafts are of more concern, such as aircraft maintenance, aircraft routing and crew assignment (for an overview, see [15, 61] and the literature cited therein). Furthermore, keeping an aircraft on the ground is expensive. Hence, it might be cheaper to reassign passengers to concurring airline companies rather than delaying an aircraft. However, optimization has been used also to address the
connections between aircrafts. In general, these approaches aim at a priori changing the arrival and departure times of the flights a little bit within a small time window, such as to minimize the expected number of missed connections. Hence, a probability distribution for the expected delays of each aircraft is assumed as given, and is in general computed by analyzing the actual landing and take-off times of the aircrafts in a given schedule. This problem was most recently addressed in [39], where it is modeled as an ILP and solved using column generation and branching strategies.

The results in the area of delay management show that the problem has been mainly addressed from an operational point of view rather than from a combinatorial and theoretical point of view. The reason for this fact is that planners are interested in models capturing most aspects of the real-world problem at hand, and in approaches delivering a (suboptimal) solution quickly which can be used as a starting block, or which deliver an evaluation of a proposed policy. In this thesis, we strip the delay management problem of most operational side constraints with the goal of understanding one combinatorial aspect of delay management.

### 2.2 A general model

In this section, we first describe a general model for delay management. In the next section, we describe the specific models used for addressing the complexity of delay management within this thesis.

Our interest for delay management is mainly theoretical; thus, the analyzed models introduce many restrictions and assumptions, which we discuss throughout the section and in Sections 2.3 and 2.4. However, the resulting models are unsuited for practical applications. Our theoretical analysis is aimed at understanding the structure of the resulting model, and provide some insight on what aspects of delay management should be understood to handle the problem in practice. Our simplified models allow to pinpoint some of these aspects.

To specify the general problem precisely, we introduce some notation. However, a generous portion of this notation falls away on our specific models as a consequence of the introduced restrictions.

Our model is a variant of an event-activity network, a graph representation used for representing a railway system. Originally in-
introduced for railway networks in [45], they are commonly used in timetabling (e.g., [48]) and periodic scheduling [57] and has also been used, in different variants, for modeling delay management [55]. In an effort to understand theoretical aspects of delay management, our model is simpler than the classical event-activity network for delay management. Due to these simplifications, specific practically relevant characteristics cannot be modeled directly, even though they can be modeled with the classical event-activity network.

**Trains, stations, services and intermediate stops** In our model, a train carries out a service between stations. We specify what we mean by this hereafter. By *train*, we mean the physical mean of transportation that travels on railway tracks, built by an engine (locomotive) and some cars for the passengers. Passengers are said to *board* a train and *alight* from a train. A *station* is a physical place where a train can stop. Passengers may board or alight from a train *stopping* at a station. By *service* of a train (short: *train service*) we mean the act of traveling, at a specific time, from a source station to a terminal station. A *leg of a service of a train* (short: *leg*) is a direct connection between two different stations served by a train, namely the leg’s source station and the leg’s terminal station. Thus, each leg is associated to exactly one train and train service; each train service is built by one or more legs, and in turn the train service defines the times the train leaves and reaches its leg’s source and terminal stations, respectively. Passengers can board or alight from a leg only at the leg’s source and terminal stations, and the train serving the leg does not stop between these stations. Considering a fixed station, the legs having as terminal that station are called *inbound*, the legs having as source that station *outbound*. A train service is built by a non-empty ordered list of consecutive legs. By *consecutive*, we mean that the terminal station of a leg in the list coincides with the source station of the subsequent leg in the list. A train having a service built by more than one leg stops at all stations used by the legs. During these so-called *intermediate stops* of the train service at such stations, passengers can alight from inbound legs or board outbound legs involving the station. If the timing is right (we will specify what we mean by this later), passengers can *connect* between legs of different train services by alighting from an inbound leg and boarding an outbound leg served by a different train at stations where the train service stops. Passengers are said to *transfer* from a leg served by a train to a leg served by a different
train. We formalize these notions in the railway network.

**The railway network** We represent the railway network as a directed acyclic graph $G = (V, E)$, sketched in Figure 2.2.1, and specified as follows. The graph is a time expanded representation of the services of the trains. Each node $v \in V$ corresponds to a station at a specific time $t_v$; each edge $e \in E$ represents a leg of a service of a train. This representation and the definitions above imply several key aspects. First, each edge is associated to exactly one train service; thus, it is also uniquely associated to a train, and to the time the train operates the service; given a leg $e = (u, v) \in E$, we refer to the departure time of the train serving leg $e$ from station $u$ and to the arrival time at station $v$ of the train serving that leg as $t^u_e$ and $t^v_e$, respectively. Naturally, the time expanded graph representation requires time consistency; hence, $\forall e = (u, v) \in E : t_v \geq t^v_e \land t_u \leq t^u_e$, and $t^u_e < t^v_e$. Second, a leg involves two physically different train stations. Edges between different time representations of the same station are not allowed. Thus, two consecutive legs served by the same train are incident to the same node of the graph.

The service of a train is represented by an ordered list of legs $r = \{e_1, \ldots, e_j\}$ and corresponds to a directed (graph theoretic) path in the time expanded graph $G$. The path is simple, as time expansion makes $G$ acyclic. We refer to the set of train services as $\mathcal{R}$. We stress that the legs served by two different train services $r_1, r_2 \in \mathcal{R}$ are disjoint for all possible pairs of train services, i.e., $r_1 \cap r_2 = \emptyset$, and that the set of edges $E$ of the graph is specified by the disjoint union of the legs served by the trains: $E = \bigcup_{r \in \mathcal{R}} r$. In our model, we assume that each physical train is used for exactly one train service. We assume this for the following reason: should a train run off-schedule, this might have an impact on another service served by that train. Modeling this possibility within delay management requires a higher level of detail, which we want to avoid in this setting. This assumption can nevertheless be interpreted as the operator introducing a big time safety margin between the services carried out by the same train.

**Passengers and transfers** The customers of the railway network are the passengers. Passengers wish to travel from an origin to a destination station. To do so, they use the services provided by the trains. A passenger may use the service provided by one or more trains to
reach her destination, and will thus connect between trains, given that connections exist between the trains she wishes to use. In our model, a passenger travels between two different nodes in $V$. In order to accomplish that, she uses the services provided by the railway network. In this context, several restrictions apply, and are discussed in the following. A passenger may use more than one leg to travel. If she wishes to transfer from a leg $e_i$ to a leg $e_j$ served by different trains, we require the legs to be consecutive (as defined above). Thus, the terminal node $v$ of $e_i$ is equal to the source node of $e_j$. This restriction prevents a passenger from remaining at the (physical) station corresponding to the node $v$ for some time and catch a train departing (much) later. However, the act of waiting at a station could be simulated by introducing a leg between the two nodes of the same station. By adding a slack time on this leg, it effectively simulates the act of waiting, as edges with slack times intuitively act as buffers for delays (we introduce this concept later, in Section 2.3.3, since we will generally not consider slack times). Note that edges representing the passenger’s act of waiting at stations exist in the general event-activity network.
2.2. A general model

Thus, we restrict passengers to travel along legs which build a directed path in the network $G$. We represent the passenger’s travel intention as passenger paths, defined as follows. A passenger path $P = \{e_1, \cdots, e_\ell\}$ (see Figure 2.2.1 for an example) consists of an ordered set of consecutive edges of the railway network, which form a directed path in $G$. These edges represent the different legs the passenger (or more than one) wishes to travel consecutively from her origin node $s(P) \in V$ to her destination node $d(P) \in V$. The legs of a passenger path are fixed, and can be seen as the train services preferred by a passenger to travel between the two nodes. In particular, passengers cannot dynamically adapt the choice of the traveled legs, possibly as a reaction to trains not running as planned (we specify the exact nature of delays in the following). For example, this inability may be a consequence of passengers being unable to obtain up-to-date information on delays. Naturally, fixing the passenger paths greatly simplifies the model and the actions which can be undertaken to handle delays. We briefly discuss the variant where passengers can adapt their route as a reaction to the delays at the end of this chapter; the obtained results also justify our choice of this simpler model. Clearly, the edges building a passenger path need not be all part of the same train. Whenever two consecutive edges $(u, v)$ and $(v, w)$ in a passenger path are served by different trains, the passengers must connect between the legs at the node $v$. We allow multiple passenger paths between the same origin-destination pair of nodes; these different passenger paths reflect the different choices of passengers on how to travel between these origin-destination nodes, given that there are several possibilities on how to travel.

The set of all passenger paths is represented by a set $\mathcal{P}$. Finally, each passenger path $P \in \mathcal{P}$ has an associated weight $w(P) \in \mathbb{N}$. This weight can represent the number of passengers using that passenger path, or the importance of a passenger path in a more abstract sense. For instance, one could weigh a passenger path commonly used by lawyers to get to work more than a passenger path used by students to get to school, since the latter will probably not sue you.

**Delays** Rather than associating delays to trains, we associate externally induced delays to passenger paths. This choice has several advantages, as we can model various delay causes: not only can we model trains running off schedule (such as delayed inbound legs), but
also delays caused by last-minute platform changes (thus passengers must walk to the new platform at the last minute, getting delayed. By considering a snapshot of this situation, these passenger paths result in starting at that station with a source delay). The connection between the model where delays are defined on passenger paths and a model where delays are defined on trains is discussed in Section 2.4.2.

Due to this modeling aspect, the choice to be made is whether a leg should wait for a delayed passenger path, or not. We define the source delays as a function $D : P \rightarrow \mathbb{Q}_0^+$, given as part of the input of an instance. We refer to the passenger paths as source delayed or source punctual, depending on whether they have a (strictly) positive source delay $D(P) > 0$ or not.

**Delays and transfers** A source delayed passenger path $P$ can only board its first leg if the departure time of that leg is delayed by waiting for $D(P)$ time units. Thus, if a leg $e = (u, v)$ waits for $\delta \in \mathbb{Q}_0^+$ time units, it updates its departure time $t_{e}^u$ to $t_{e}^u = t_{e}^u + \delta$. When not ambiguous, we occasionally say that a train service waits at a station. If the leg waits for a delayed passenger path, it will maintain that delay up to its terminal station (i.e., $\tilde{t}_e^v = t_e^v + \delta$), and propagate that delay to all consecutive legs of the train service it is part of. Furthermore, this delay is also propagated to the passenger paths which board that leg. In general, we assume that trains are unable to catch up on the accumulated delays while traveling their legs, or while stopping at stations. This restriction keeps the model simple, allowing us to focus on delay propagation, and can be seen as reflecting a very tight schedule run by an operator. We describe a more realistic scenario where trains are able to catch up a specific amount of time per leg later in this section, and also provide some complexity results for this model in Chapter 3.5; however, our basic model will disregard this aspect. Thus, if the service of a train is given as $r = \{e_1, e_2, e_3, e_4, e_5\}$, and the leg $e_3$ waits for $\delta$ time units, the delay $\delta$ is propagated to the legs $e_4$ and $e_5$, which we also say to be waiting.

**Delays and passenger paths** After having alighted from a leg that waited for $\delta$ time units, a passenger path may only transfer to its next leg served by a different train if the transfer is possible with respect to the times of service. The exact specification on when transfers between consecutive waiting legs are possible is given in the next
section, when we introduce the restricted model we generally use. Intuitively, a transfer from a leg \( e_i = (u, v) \) which waits \( \delta \) time units to a consecutive leg \( e_j = (v, w) \) served by a different train is only possible if the departure time \( t^v_{e_j} \) gives the delayed passenger paths sufficient time to board, or is adapted to do so.

If a transfer is not possible, the passenger path misses that connection. A passenger path missing a connection is said to be dropped. We also say that a passenger path is dropped if it is source delayed and its first leg does not wait. We refer to passenger paths not missing a connection as maintained paths.

If a passenger path is dropped, the passenger travels as follows. We point out that the new travel intention cannot be modeled directly in our model, but that for our purposes and some modeling assumption introduced throughout this chapter this fact is irrelevant. The passenger path chooses the services provided by trains passing through the exact same stations as the planned one, but now shifted later in time. Of the train services available later, the passenger path chooses the one bringing it to its destination station as early as possible, considering the current passenger path’s delay. This choice can be seen as a passenger still traveling along the same stations as initially planned, and using the next (in time) provided service. This assumption is especially reasonable when considering periodic timetables, that is, timetables where each train service is planned to be repeated after a fixed amount of time (the so-called period of the timetable). The time difference between the actual time of arrival of a passenger path \( P = \{e_1, \cdots, e_\ell\} \) at the station identified by the node \( d(P) \) and the planned time of arrival \( t^d_{e_\ell}(P) \) is called the arrival delay of the passenger path \( P \).

**Objective function and solution** Given this model, we remain with specifying the optimization goal we are seeking for. Intuitively speaking, we would like to keep the discomfort faced by passengers small, despite the given source delays. We do this by minimizing, over all passenger paths, the weighted sum of each passenger path’s arrival delay. For the time being, we omit a clearer definition of our objective function, which is discussed in detail in the next section.

A solution to the problem is a delay policy. Given an instance of the delay management problem (built by a graph, the passenger paths, the source delays and the weights of the passenger paths, the
train services and a compact representation of the delay faced by each passenger path for every delay policy), a delay policy specifies which legs wait and for how long. Naturally, the delay policy must be feasible with respect to the modeling assumptions introduced above, and in particular be consistent for the legs of a train service, i.e., once a leg of a train service waits, all subsequent legs of the same train service must also wait. Now, we see that the previously defined arrival delays of an instance’s passenger paths are a function of the chosen delay policy for that instance.

2.3 Models within this thesis

The previously presented model disregards many aspects which are relevant at an operational level within a railway system, and introduces restrictions which are unrealistic (as, for instance, the fact that transfer between legs are only possible if they are consecutive in the graph). Although the restrictions we have already introduced (and the restrictions we introduce in the following) make the model unsuited for reality, we again point out that we focus on this very simple model to gain some insight on delay management. Our first goal is not to solve the delay management problem in practice (although we would certainly like to), but to provide a speck of insight on what aspects make delay management difficult (or easy) to solve. With our restrictions, we focus on one aspect, namely on the decision of which passenger paths one should maintain. It is unclear if such a focus allows to derive some insight on the real-world problem; however, it points out which aspects should be carefully considered also in the practical setting. We discuss some of the omitted operational aspects in Section 2.4.

2.3.1 Restrictions common to all considered models

In this section, we introduce the restrictions common to all models we consider in this thesis, as well as the objective functions considered. The severity and the impact of these restrictions is discussed in the following.
2.3. Models within this thesis

Binary delays  A significant restriction addresses the source delays, which we restrict to be binary. Thus, a passenger path $P \in \mathcal{P}$ is either source punctual (and thus have source delay $\mathcal{D}(P) = 0$) or source delayed, in which case it has a fixed, constant delay of $\delta$ time units, the same for each source delayed passenger path (thus, $\mathcal{D}(P) = \delta$). This restriction simplifies the problem considerably, as will become clear in the next few pages.

Instantaneous transfers  A further simplification is to assume instantaneous transfers. Thus, it takes no time for the passenger paths to transfer, which means they can transfer between leg $e = (u, v)$ and leg $f = (v, w)$ of different train services if $t_v^v \geq t_v^u$. This assumption alone is not too far fetched. Indeed, a uniform, station-dependent transfer time can be added to the inbound leg’s arrival times; considering uniform transfer times is often reasonable, since transfer times are only estimates, and different passengers take differently long to transfer (the fit teenager can run to catch a train, the young mother with her two small children cannot).

We make another crucial assumption. We discuss the impact of all assumptions on the model later; however, we immediately comment on this one. We assume that for each leg $e = (u, v)$, the arrival time $t_v^e$ at node $v$ coincides with the node’s time $t_v$; similarly, we assume that the departure time $t_u^e$ from node $u$ coincides with the node’s time $t_u$. Thus, $\forall e = (u, v) \in \mathcal{E}$: $t_v^e = t_v \land t_u^e = t_u$. This restriction prevents a train service from catching up on some of its delay at a station and propagates the delays entirely also to connecting legs that wait. Stations are actually the (physical) place where time buffers preventing the propagation of delays are very likely to be introduced by planners. Also in this case, the general event-activity network allows for this possibility. In our model, waiting at stations can again be simulated by introducing legs with slack times between two different time representations of the same station. In general, we stick to the version without slack times for simplicity, and prevent any possibility of a train service to reduce its delay. This restriction also allows to neglect the timing information when considering transfers, since arrivals, transfers and departures take place concurrently. The exact times are irrelevant: only the information whether the inbound legs run as scheduled or not matters, as the delay of off-schedule legs is clearly of size $\delta$ (we review this aspect later). Implicitly, time is still
present, and associated to the node. The direction of the edges give a relative order of the times of the nodes. In an offline setting we can neglect exact times, as we only have to decide which passenger paths we want to maintain. Naturally, a timeless environment is artificial in a railway environment. Our restrictions further void the possibility of saying which of two timetables is more robust. Although such a setting can be seen as a very, very special special-case of operation, our goal is to simplify the model sufficiently to allow some understanding of the whole delay management problem. However, the restriction we impose might turn out to be so severe that our results do not apply to the real-life problem.

**Periodic timetable and delay propagation** Often, railway systems run according to a periodic timetable. Hence, the same sequence of stations is visited by different train services every, say, half hour or every hour. The period is in general not the same for all train services, and depends on the type of service provided (commuter trains tend to have shorter period than intercity trains) or on the importance of the served stations, and the number of passengers using the service. In general, we assume that the delay $\delta$ is small compared to the time period. The reason for this is twofold. On the one hand, if the delay is of similar size to the time period, it makes little sense within our context to make a leg wait for a delayed passenger path, as the next scheduled train services will bring the passenger path to its destination shortly after anyways. The second reason is that delay management generally focuses on small delays, which can be addressed by changing the original timetable by not too much.

In our model, we assume that train services travel according to a periodic timetable with period $T$, equal for all train services. Although this assumption can be interpreted as considering only one specific service provided by the railway operator (for instance, we could focus on the intercity train service), we point out that this restriction is mainly introduced for simplicity. Now, if a passenger path misses a connection, it can board the same sequence of legs of the next time period. Recall that dropped passenger paths choose the earliest possible train service.

Furthermore, we make the strong assumption that delays do not propagate to later periods of the timetable. Thus, should a passenger path miss a connection, it will not encounter *any* further delays
and arrive at its destination as scheduled within the next period of the timetable. Naturally, examples exist where delays do propagate or where a passenger path is delayed again because of another source delayed passenger path; we do not consider them as matter of simplicity, since this assumption allows us to easily determine the arrival delay of dropped passenger paths (see next paragraph). Generally, combinatorial approaches to delay management tend to make this artificial assumption [54, 55]. Indeed, although being a very interesting and realistic setting, it is unclear how to handle delay propagation to the next period of the timetable when representing the passenger’s travel intentions as origin-destination pairs of stations in the network. This even holds for IP formulations; considering it requires keeping track of a passenger path even when it has been dropped, thus requiring a representation (explicit or implicit) of the legs used by a passenger path when it is dropped (and note that passenger paths could be dropped again in subsequent periods of the timetable).

**Implications** The introduced restrictions have several effects. As stated, the combination of binary delays and instantaneous transfers simplify the question addressed in an optimization. Indeed, the decision to be taken when confronted with a delayed passenger path (which may be source delayed or having been delayed as a consequence of a leg waiting) is only whether to wait, or to depart as scheduled. If a leg waits, the waiting time is clear: as the only magnitude of delay present is $\delta$, there is no possibility to catch up on any acquired delay, and there is no additional time at stations, if a train decides to wait it will wait for $\delta$ time units. Furthermore, a delayed passenger path may only board a leg if that leg decides to wait for it for $\delta$ time units.

In our model, a train service cannot accumulate more than $\delta$ delay along its legs, as this would require that a passenger path exists which has a source delay larger than $\delta$. Therefore, a train service $r = \{e_1, \ldots e_\ell\}$ which waits at a leg $e_j \in r$ allows all delayed passengers paths intending to board at legs $e_k, j \leq k \leq \ell$. Thus, no passenger path reaching a leg $e_k, j \leq k \leq \ell$ of $r$ with a delay can miss the connection involving the train service $r$, given that $e_j$ waits.

Let $\mathcal{W}$ be a function that for a given instance of delay management, built by a graph $G$, the passenger paths $\mathcal{P}$ with weights $w$ and source delays $\mathcal{D}$, the train services $\mathcal{R}$ and the period $T$ returns the
Chapter 2. A model for delay management

set \( \Pi \) of feasible delay policies for the given instance. With the restrictions above, each delay policy \( \pi \in \Pi \) for a specific instance only needs to specify which legs wait. Hence, a delay policy can be seen as a vector of boolean values, one for each leg (with the restriction that the decisions for the legs within a train service are consistent); a value true for a leg means that the leg should wait, the value false that it should depart as scheduled. Let \( \mathcal{A} : \Pi \times \mathcal{P} \to \{0, \delta, T\} \) be a function which, for a specified delay policy \( \pi \in \Pi \) and for each passenger path \( P \in \mathcal{P} \) on the network \( G \) returns the arrival delay of \( P \) with policy \( \pi \). A source delayed passenger path \( P \) is maintained only if the delay policy makes all the legs the passenger path wishes to board wait; in this case, the passenger path has an arrival delay of \( \mathcal{A}(\pi, P) = \delta \) time units. A passenger path boarding a leg that waits is maintained only if all the consecutive legs the passenger path wishes to board from that point on wait with the given delay policy. Also in this case, the passenger path has an arrival delay of \( \mathcal{A}(\pi, P) = \delta \) time units. A passenger path \( P = \{e_1, \cdots, e_\ell\} \) is dropped if the delay policy specifies that a leg \( e_i \in P \) waits and that a leg \( e_j, j > i \) departs as scheduled. Note that for this to be possible, boarding \( e_j \) involves a passenger transfer between two (physically) different trains. If dropped, the passenger path boards the same train services within the next period, and arrives at its destination with an arrival delay of \( \mathcal{A}(\pi, P) = T \) time units. Only a source punctual passenger path \( P \) can arrive at its destination with arrival delay equal to zero (thus, \( \mathcal{A}(\pi, P) = 0 \)). This only happens if none of the legs it boards wait in \( \pi \).

**Objective function** We wish to find a delay policy \( \pi^* \) that minimizes the total passenger delay defined as the weighted sum of the arrival delays of the passenger paths, shortly discussed in the following. Here, the arrival delay of a passenger path is defined as above, and each passenger path’s arrival delay is multiplied with the passenger path’s weight. Thus, we have the following objective function:

\[
\min_{\pi \in \Pi} \sum_{P \in \mathcal{P}} w(P) \cdot \mathcal{A}(\pi, P).
\]

This objective also includes the source delay of the passenger paths, which in this setting cannot be optimized: Indeed, the best policy for any source delayed passenger path induces an arrival delay of \( \delta \) to it. In this thesis, we also address the problem variant
where the source delays are not counted: thus, in the objective function the source delay of a passenger path is subtracted from its arrival delay. We refer to this second objective as the *additional passenger delay* objective. Formally, in this case we wish to find a delay policy $\pi^* \in \Pi$ minimizing the objective function

$$\min_{\pi \in \Pi} \sum_{P \in \mathcal{P}} w(P) \cdot (A(\pi, P) - D(P)).$$

We use the latter objective function mainly to address inapproximability results and in the context of lower bounds for online algorithms.

The choice of the total passenger delay objective might seem, at first sight, quite arbitrary. However, by choosing a trade-off between delaying some source punctual passenger paths to allow transfers of delayed passengers and dropping some passenger paths the objective reflects a customer-oriented view: we aim at maintaining some connections for delayed passenger paths, without penalizing the rest of the passenger paths too much. In the following, we shortly discuss some other reasonable objective functions. It turns out that they can trivially be optimized in our model.

Ideally, we would like to keep the delay for all passengers small. Therefore, we can aim at minimizing the maximal arrival delay of all passenger paths. In our model, this objective results in making legs serving as connections to delayed passenger paths wait, as in this way the maximal delay is $\delta < T$ for each passenger path. A similar situation arises if we seek at minimizing the number of missed connections. In this case, a trivial optimal solution is to delay *all* legs in the network. In an opposite way of thinking, another objective reflecting a good quality of service for passengers is to maximize the number of punctual passengers paths. Such an objective would however always result in dropping all source delayed passenger paths.

### 2.3.2 Problem statements

In this section, we formally define the problems which we address. We start with the more general variant.
Chapter 2. A model for delay management

**Definition:** Binary delay management problem with intermediate stops

**Instance:** A railway network \( G = (V, E) \) with instantaneous transfers, a set of passenger paths \( P \), the binary source delays \( \mathcal{D} \) and the weight function \( w \) of the passenger paths, the set of train services \( \mathcal{R} \), and the period \( T \).

**Goal:** Find a delay policy \( \pi \in \Pi \), which specifies which legs wait for delayed passenger paths, and minimizes the total passenger delay.

We address the above problem only marginally. The problem variant we analyze in most detail is as follows.

**Definition:** Binary delay management problem

**Instance:** A railway network \( G = (V, E) \) with instantaneous transfers, a set of passenger paths \( P \), the binary source delays \( \mathcal{D} \) and the weight function \( w \) of the passenger paths, the set of train services \( \mathcal{R} \) with each train service built by exactly one leg, and the period \( T \).

**Goal:** Find a delay policy \( \pi \in \Pi \), which specifies which legs wait for delayed passenger paths, and minimizes the total passenger delay.

Here, we make the additional restriction that each train service is built by exactly one leg. Thus, there is a one to one correspondence between train services and legs, and each leg corresponds to exactly one physical train. This restriction keeps the problem simple, as it is not necessary to ensure any consistency in the delay policy anymore: indeed, all possible \( 2^{|E|} \) wait / non-wait assignments to the legs are feasible delay policies. We discuss this restriction in some more detail in Section 2.4.1, where we show how an instance with intermediate stops can be mapped to an instance without intermediate stops in polynomial time, by introducing a polynomial number of additional legs and passenger paths. The mapping introduces passenger paths of heavy weight; these passenger paths enforce the needed consistency within a train service in all optimal solutions. Thus, as long as we are looking for exact algorithms, the restriction that train services are built by exactly one leg is not limiting.
2.3.3 Delay management with slack times

We also briefly address the setting where trains can catch up on their delay, for example by driving faster than scheduled. We call the possible amount of time by which each leg \( e = (u, v) \in E \) can reduce its delay the *slack time* of that leg, denoted by \( S(e) \), where \( S(e) \in \mathbb{Q}_0^+ \). Note that this slack time is a function of the used leg, thus also of the actual train being used for serving the leg. Hence, we can model train-composition and track dependent slack times combined.

Trains must not arrive earlier than scheduled, so a leg’s slack time can only be used if the leg waits, and the maximum amount of slack time which can be used is, in view of the previous restrictions, upper-bounded by \( \delta \) time units per leg. Preventing trains from traveling earlier than scheduled is common practice in railway operations, as allowing this requires to modify the planned operations unnecessarily and, even worse, it would make punctual customers missing a train traveling earlier than scheduled extremely angry. Given a slack time of \( \delta \) time units, a leg can wait for delayed passenger paths and still arrive at its target station as scheduled. Thus, it is never harmful for a leg to exhaust all available slack time on a leg, since doing so has either no effect (as the next leg of the train service can always wait) or decreases the arrival delay of some passenger path. Here, we do not specify the arrival delay of the passenger paths formally. The occurring arrival delays will be specified in the particular settings of the delay management problem with slack times we analyze.

Within this context, a delay policy specifies which trains wait; again, the goal is to minimize the total passenger delay.

We address the following two problem variants with slack times. Recall that the slack times defined above apply on top of the restrictions common to all problem variants of Section 2.3.1.
Chapter 2. A model for delay management

Definition: **Binary delay management problem with slack times and intermediate stops**

Instance: A railway network \( G = (V, E) \) with instantaneous transfers, a set of passenger paths \( \mathcal{P} \), the binary source delays \( \mathcal{D} \) and the weight \( w \) of the passenger paths, the set of train services \( \mathcal{R} \), the period \( T \) and the slack times \( S \) of the legs.

Goal: Find a delay policy \( \pi \in \Pi \) which specifies which legs wait for delayed passenger paths and minimizes the total passenger delay.

This variant is just the binary delay management problem with intermediate stops with the additions of the slack times.

Definition: **Binary delay management problem with slack times and without intermediate stops**

Instance: A railway network \( G = (V, E) \) with instantaneous transfers, a set of passenger paths \( \mathcal{P} \), the binary source delays \( \mathcal{D} \) and the weight \( w \) of the passenger paths, the set of train services \( \mathcal{R} \) with each train service built by exactly one leg, the period \( T \) and the slack time \( S \) associated to each leg.

Goal: Find a delay policy \( \pi \in \Pi \), which specifies which legs wait for delayed passenger paths and minimizes the total passenger delay.

This second variant is basically the binary delay management problem (thus without intermediate stops) with the inclusion of slack times. Thus, we again assume that each train service serves exactly one leg.

2.4 Restrictions, equivalences and a first reduction

The model described in the previous section is obviously very restricted and disregards many relevant constraints of every-day railway planning. To name a few, we do not consider the following important aspects: the capacities of trains, which are bounded, thus rendering some waiting policies unfeasible; the availability of platforms at the stations where trains wait; the availability of tracks for trains running
2.4. Restrictions, equivalences and a first reduction

off-schedule; the headway constraints; the rolling stock availability; crew scheduling. All the previous aspects play a crucial role in a smooth operation of railways, and are influenced by unpredictable and scheduled deviations from the original timetable. The goal of this thesis is to give insight on the combinatorial structure of delay management. We decided to focus on just one aspect of the delay management problem; we chose to address the question of delay propagation, that is, the question on which trains should wait for delayed passengers to improve the quality of service for passengers with respect to connections when trains are late. To that aim, we disregard the aspects mentioned above, which do not directly influence this decision, but rather constrain them from an operational point of view.

Considering our focus, we address three relevant restrictions in our model for delay management in the following sections. First, we address the lack of intermediate stops of trains. Then, we discuss the impact of defining input delays on passenger paths rather than on legs, which seems more natural. Finally, we address the possibility of a railway operator to reroute passengers as an additional measure to keep the arrival delays small. By rerouting passengers paths we mean the possibility of choosing the legs used by the passengers paths to travel from their origin node to their destination node; combined with the choice of a suited delay policy, this possibility allows passenger paths to travel through legs causing them the least arrival delay.

However, before addressing these key aspects, there are two additional restrictions that may have jumped to the eye of the reader. First, we restrict delays to be binary and the transfers to be instantaneous on legs with very special times; second, we generally assume that trains cannot catch up on their delay. There are several reasons for these restrictions, but the general motivation is that we keep the problem as simple as possible to understand one aspect of delay management. Multiple sized delays only make the problem harder, both from a combinatorial point of view and from an understanding point of view. The same holds for slack times. Moreover, the key point in delay management we aim at understanding is what happens when delays propagate through the network. In our model, propagation is inevitable as soon as some leg waits for the delayed passengers paths, and this propagation is only stopped by dropping transferring passenger paths or if the passenger paths on board of a leg that waits do not transfer further. Thus, even with these restrictions, one crucial aspect
of delay management is captured, namely the propagation of the delays in the railway network, and we are focusing on this aspect. We show the impact of allowing slack times in Section 3.5. The results point out the added complexity introduced by slack times.

Although the resulting model is artificial, the lack of understanding of the more complex event-activity network from a computational complexity point of view serves as motivation for simplifying the model considerably. The simplifications allowed us to focus on a single aspect, and derive interesting complexity results, which show that the aspect of deciding which trains should wait is by itself computationally difficult (see Section 3.1).

### 2.4.1 Intermediate stops

In general, we do not consider trains with intermediate stops in our offline setting. Hence, we generally assume that each train service is built by exactly one leg, and thus by a single edge of the graph. Again, this restriction aims at keeping the structure of the problem simple, but still captures the issue of the propagation of delays through the network when trains wait to guarantee a connection. Since knowing the exact leg at which a train service begins to wait is of crucial importance to compute the arrival delays of the paths, a simple substitution or contraction of the legs of a train service is not sufficient to remove intermediate stops. In this section, we show a polynomial-time transformation from the binary delay management problem with intermediate stops to the binary delay management problem. Thus, every instance of the first problem can be transformed into an equivalent instance of the latter problem. This reduction does not maintain approximation ratios across the models, as it introduces many source delayed passenger paths of high weight. As long as we are interested in exact algorithms, the offset introduced into the total passenger delay objective by these source delayed passenger paths does not matter.

Clearly, the binary delay management problem is a special case of the binary delay management problem with intermediate stops. We show next how to map an instance $I$ of the binary delay management problem with intermediate stops into an instance $J(I)$ of the binary delay management problem (without intermediate stops).

The general idea of the reduction is to interpret every edge in $I$ as a train service in $J$. To ensure the consistency required by train
services in $I$, we introduce passenger paths of high weight which use two consecutive legs of a train service; it will be possible to delay these passenger paths, but not to drop them, thus guaranteeing that all legs of a train service which follow a leg that waits must also wait.

For the instance $I$ of the binary delay management problem with intermediate stops, let $G$ be the railway network, $\mathcal{P}$ the passenger paths, $\mathcal{D}$ the source delays of the passenger paths, $w$ the weights of the passenger paths, $\mathcal{R}$ the set of train services and $T$ the period of the time table. For the instance $J(I)$ we construct, we use an apostrophe to refer to the equivalent input parts. Thus, the railway network is $G'$, the set of passenger paths $\mathcal{P}'$, and so on.

The instance $J(I)$ uses the same period of the timetable as $I$: $T' = T$. We build the network $G'$ by extending $G$, and consider each edge $e \in E$ of $G$ as train service $r'_e = \{e\}$ of $J(I)$. Thus, $\mathcal{R}' = \{r'_e | \forall e \in E : r'_e = \{e\}\}$, and each train service of $\mathcal{R}'$ has no intermediate stops. Thus, there is a bijection between the edges in $G$ and this initial set of train services $\mathcal{R}'$ in $J(I)$. Next, we introduce all passenger paths $\mathcal{P}$ into $\mathcal{P}'$, and set the same weights and source delays for these paths in $J(I)$. Thus, a path in $P \in \mathcal{P}$ uses the corresponding legs of $G$ in $G'$, has weight $w'(P) = w(P)$ and source delay $\mathcal{D}'(P) = \mathcal{D}(P)$. For each train service $r \in \mathcal{R}$, we introduce a new node $v_r$ in $G'$. For each edge $e_i = (v_i, v_{i+1}), i \in \{2, \cdots, j\}$ of a train service $r = \{e_1, e_2, \cdots, e_j\} \in \mathcal{R}$, we introduce an edge $e'_i = (v_{i+1}, v_r)$ in $G'$ and the train service $r'_{e_i}$ into $\mathcal{R}'$. For each such edge $e'_i$, we introduce two passenger paths into $\mathcal{P}'$: a source punctual passenger path $p'_i = \{e_{i-1}, e_i, e'_i\}$ of weight $w'(p'_i) = 1 + T \sum_{p \in \mathcal{P} : \mathcal{D}(p) = \delta} w(p)$, and a source delayed passenger path $q'_i = \{e'_i\}$ of weight $w'(q'_i) = w'(p'_i)$. The goal of these passenger paths is as follows: the high weight $w'(p'_i)$ guarantees that $p'_i$ is maintained in every optimal solution, as it is cheaper to drop all source delayed passenger paths coming from $I$ than dropping $p'_i$. The heavy weight of the source delayed passenger path $q'_i$ guarantees that it is cheaper to delay $p'_i$ by making $e'_i$ wait than to drop $q'_i$. Thus, in every optimal solution to $J(I)$, $p'_i$ is delayed as a consequence of $e'_i$ waiting for $q'_i$. Because of this, $p'_i$ has an arrival delay $\delta$, and the legs $e_i$ and $e_{i-1}$ can wait causing no additional cost for $p'_i$. All considered, $p'_i$ can be delayed but not dropped, ensuring the necessary consistency for the considered train service $r$.

We get the following theorem:
Theorem 2.1. The minimum cost delay policy for minimizing the total passenger delay of any instance \( I \) of the binary delay management problem with intermediate stops can be derived by efficiently mapping \( I \) to an instance \( J(I) \) of the binary delay management problem and by minimizing the total passenger delay of \( J(I) \).

Proof. We have already described the transformation, which introduces, in polynomial time, a polynomial number of nodes, edges and passenger paths. We are left to prove the correctness of the theorem formally. Let \( \text{OPT}(I) \) be the optimal objective value for \( I \), and assume \( I \) has \( \rho = \sum_{r \in R} (|r| - 1) \) intermediate stops, where by \(|r|\) we mean the number of legs served by the train service \( r \). Now, we claim that \( I \) has an optimal solution of cost \( \text{OPT}(I) \) if and only if \( J(I) \) has an optimal solution of cost \( \text{OPT}(J(I)) = \text{OPT}(I) + 2\rho\delta(1 + T\sum_{p \in P} w(p)) \). By proving this claim, we prove the theorem.

Given an optimal delay policy for \( I \) of cost \( \text{OPT}(I) \), it is easy to construct a delay policy of cost \( \text{OPT}(I) + 2\rho\delta(1 + T\sum_{p \in P} w(p)) \) for \( J(I) \): we simply apply \( I \)'s optimal delay policy for the edges in \( G \) to the corresponding edges of \( G' \), and make all the edges in \( G' \) of type \( e'_i \) wait. This delay policy causes the same delay to the passenger paths coming from \( \mathcal{P} \), and obviously a delay of \( \delta \) to the passenger paths of type \( q'_i \). It remains to be shown that the passenger paths of type \( p'_i \) are not dropped. Since each of these passenger paths uses legs within a train service (where no drop can take place) and ends with an edge \( e'_i \) that waits, these passenger paths are never dropped in a feasible delay policy for \( I \) and thus arrive at destination with a delay of \( \delta \). Overall, the constructed delay policy for \( J(I) \) has the stated objective value.

For the other direction, we first focus on the interaction between the passenger paths \( p'_i \) and \( q'_i \) and the passenger paths from \( \mathcal{P} \). All optimal delay policies maintain the passenger paths of type \( p'_i \), since it is cheaper to drop all source delayed passenger paths from \( \mathcal{P} \) than dropping one passenger path \( p'_i \). Moreover, all passenger paths of type \( q'_i \) are maintained, as it is cheaper to delay \( p'_i \) than drop the source delayed passenger path \( q'_i \), and each passenger path \( q'_i \) interacts exclusively with one passenger path \( p'_i \). Therefore, these passenger paths alone contribute with \( 2\rho\delta(1 + T\sum_{p \in P} w(p)) \) to the objective value of any optimal delay policy. Given a delay policy for \( J(I) \) of cost \( \text{OPT}(J) \), we focus on the edges of \( G' = (V', E') \) deriving from \( G = (V, E) \) and on the passenger paths stemming from \( \mathcal{P} \). Restricted on \( E \) and \( \mathcal{P} \), the delay policy causes a delay
of $\text{OPT}(J(I)) - 2\rho \delta (1 + T \sum_{p \in P} w(p))$. It remains to be shown that the policy is feasible for $I$. As previously stated, the passenger paths $p^r_i$ ensure that no drop is possible between consecutive legs stemming from a train service $r$. Finally, since such a passenger path is eventually delayed on the edge $e^r_i$, it can be delayed at no additional cost on edges from $E$. $\square$

It may appear unreasonable to introduce a separate passenger path between each pair of consecutive legs within a train service, as it is sufficient to have one additional passenger path per train service: this passenger path is built by all the legs of the train service, and ends on a dedicated edge shared with a passenger path with the same function as the passenger paths $q^r_i$. Such a construction would indeed reduce the number of passenger paths introduced, but at the cost of having passenger paths which, in the model without intermediate stops, transfer many times, i.e., at each intermediate and at the terminal station of the train service. The construction presented above keeps the number of transfers at two for each passenger path of type $p^r_i$. The hardness results and the polynomial-time approaches presented in the next chapter point out why keeping the number of transfers small is sensible.

### 2.4.2 Delayed passenger paths

In delay management, it may seem unnatural that the delays given in the input (which we call source delays) are defined on passenger paths rather than on trains. Indeed, a service provider will only decide to make some train wait for delayed passenger if the cause of the delay is within the service he provides, and drop the passengers otherwise (after all, it’s your own fault if you missed your train because you left home too late). However, modeling delays on passenger paths allows for more flexibility: as already stated, there are many causes for delays within a railway system. Defining delays on passenger paths allows to model some of these additional delay causes, without preventing us from considering delayed trains.

In this section, we sketch a reduction from the binary delay management problem, where delays are defined on passenger paths, to a model which differs from it only by the fact that source delays are defined on legs and not on passenger paths. We also sketch the other direction.
In general, we consider an instance of offline delay management as taking a snapshot of the current network situation, and looking at the source delays as they exist in the snapshot. Thus, in the setting with source delays defined on trains, all trains that at the moment of the snapshot are already known to be departing from their source station with a delay are considered source delayed. We do not consider trains departing as scheduled and arriving at their destination with a delay. Also, we consider the passenger paths to reflect the remaining travel intention at the time of the snapshot.

First, note that it is easy to construct an instance with source delayed trains from an instance of the binary delay management problem, where source delays are defined on passenger paths. For each source delayed passenger path \( P \in \mathcal{P} : D(P) = \delta \), we introduce a source delayed directed edge \( e_P \) with head in \( s(P) \) and tail in a dedicated node, and add leg \( e_P \) as the first leg of \( P \). Thus, we introduce \( O(\mathcal{P}) \) new nodes and edges in the existing network, keeping the transformation polynomial in space and in construction time. This construction introduces one additional transfer to the source delayed passenger paths. The constructed instance is equivalent to the instance of our classical model. Source punctual passenger paths are unaffected by the transformation, and source delayed passenger paths can miss what used to be (before the transformation) their initial leg also in the train-based delay setting, and do not differ from the normal model for the remaining connections.

The reduction in the opposite direction is even easier. To model a any source delayed train \( e \) in a setting where delays are modeled on passenger paths, it is sufficient to add, for each such edge, a passenger path \( P_e = \{e\} \), \( D(P_e) = \delta \), \( w(P_e) = M \), where \( M \) is a big constant. Naturally, we remove the source delay from the edge. Because of the large weight \( M \) of the passenger path \( P_e \), it will always be advantageous to make \( e \) wait for \( P_e \) rather than dropping \( P_e \). Hence, we have effectively introduced a source delay for \( e \). We remark that the sum of the weights of the original passenger paths of the instances, multiplied by the time period \( T \), is sufficient as a value for \( M \).

Thus, it is clear that these two models are equivalent, and that switching from one to the other adds only little complexity. We show later that the introduced complexity for modeling a passenger path based source delay setting through a train based source delay setting does not affect our hardness results (see Theorem \ref{thm:equivalence}).
2.4. Restrictions, equivalences and a first reduction

2.4.3 Dynamic path choices

Up to now, we have addressed only one strategy to maintain connections and to minimize delays. Indeed, making trains wait is only one option to keep delays small. Among others, another option is to reroute passengers in the network, that is, to choose the legs which passengers should use to travel in the optimization process. The new sequence of legs might thus differ from the sequence of legs passengers intended to board, thus the term rerouting. If many directed paths exist between the same origin/destination pair of nodes, this approach may allow to chose alternate routes for source punctual passengers, such that they are unaffected (with respect to the intended time of arrival) by the delay policy. In an environment where a central authority is able to gather passenger information and delay information in real time, the choice to reroute passengers in addition to letting some trains wait for the connecting late passengers may be a practicable and good option. This view is indeed not too far fetched: electronic tickets are more and more common, and in the near future it might very well be possible to gather the origin/destination pair of each passenger in real-time, and to track the exact location of each train (and thus compute the delay of a train).

In the offline view of the delay management problem considered next, the main ingredients of the problem are the same as for binary delay management problem, and include the restrictions of Section 2.3.1. We consider a railway network $G$ with instantaneous transfers operated according to a periodic timetable with period $T$; each train service is built by exactly one edge of $G$.

By allowing rerouting, the legs traveled by passengers are chosen by the delay policy. Thus, it does not make much sense to specify the sequence of legs passengers planned to travel. Thus, the travel intention of the passengers is completely specified by the node at which they want to board the first leg and by the node at which they want to alight from the last leg. This travel intention corresponds to a passenger wanting to travel starting from a physical station at a specific time, and arriving, by train services provided by the operator, at another physical station at a specified time (so, for instance, departing from Zurich HB at 16:09 and arriving in Milano Centrale at 20:35).

Thus, the travel intention $\psi$ of a passenger is represented by an origin/destination pair of nodes $s(\psi) \in V$ and $d(\psi) \in V$, the origin
node and the destination node, respectively. A route for such a pair of nodes is a sequence of consecutive legs in $G$ building a directed path from node $s(\psi)$ to node $d(\psi)$. We denote by $\Psi$ the set of all travel intentions of the passengers. We say that the service intentions have a \textit{dynamic path choice}, since they can adapt the route depending on which legs wait.

In a slight abuse of notation, $D : \Psi \to \{0, \delta\}$ specifies the source delay of a travel intention $\psi \in \Psi$. Also in this setting, we only consider binary source delays defined on passengers. Furthermore, we again associate a weight $w(\psi) \in \mathbb{N}$ to the travel intention $\psi \in \Psi$ of passengers.

A delay policy $\pi$ for this problem specifies which legs of the network wait for delayed passengers, and a set of routes, one for each source/destination pair $\psi \in \Psi$. A route for a source/destination pair $\psi \in \Psi$ specifies the sequence of legs to be traveled by $\psi$ from its origin node $s(\psi)$ to its destination node $d(\psi)$. The route causes the least possible arrival delay to $\psi$ for the given waiting decisions.

The objective we consider is to find a delay policy which minimizes the total passenger delay for the policy’s routes. The total passenger delay is defined as in Section 2.2, and the legs used by the travel intention $\psi$ is specified by the route for $\psi$ of the delay policy.

Note that it is not advantageous to split the passengers within a travel intention $\psi$ on two different routes, as they all have the same source delay and trains do not have, in our setting, a maximum capacity.

In the following, we first state the optimization problem resulting from the above definitions formally; next, we state the corresponding decision problem, and show that it is \textit{NP}-complete.

\textbf{Definition}: \textbf{Dynamic path choice delay management problem}

\textbf{Instance}: A railway network $G = (V, E)$ with instantaneous transfers, a set of service intentions $\Psi$, the binary source delays $D$ and the weight function $w$ of the service intentions, the set of train services $\mathcal{R}$, each built by a single edge of $G$, and the period $T$.

\textbf{Goal}: Find a delay policy $\pi$, which specifies which legs wait and a set of routes, one for each travel intention, which minimizes the total passenger delay.
2.4. Restrictions, equivalences and a first reduction

**Definition:** Decision dynamic path choice delay management problem

**Instance:** An instance of the dynamic path choice delay management problem, \( d \in \mathbb{N} \).

**Question:** Does a delay policy exist, such that the total passenger delay is less than or equal to \( d \)?

**Theorem 2.2.** The decision dynamic path choice delay management problem is \( \mathcal{NP} \)-complete.

**Proof.** The decision problem is in \( \mathcal{NP} \), since the weight of each route can be computed in polynomial time for any given delay policy. We show that it is \( \mathcal{NP} \)-hard by a reduction from 3-SAT [20, Problem LO2].

Given a 3-SAT formula \( C \) with \( m \) clauses \( c_j \) over \( n \) variables \( x_i \), we build the decision dynamic path choice delay management problem instance using the notation specified above as follows (see Figure 2.4.1): We set \( \delta = 1 \), \( T = 2 \), \( d = n \). The reason for these values will become clear in the following. We build the railway network \( G = (V, E) \) as follows: First, we introduce one node \( v_s \). For each variable \( x_i \in C \) we introduce four nodes, \( v_{s x_i}^s, v_{t x_i}^t, v_{x_i}^{xx_i}, v_{\neg x_i}^{xx_i} \). These nodes are interconnected through four edges \((v_{s x_i}^s, v_{x_i}^{xx_i}), (v_{x_i}^{xx_i}, v_{t x_i}^t), (v_{s x_i}^s, v_{\neg x_i}^{xx_i}), (v_{\neg x_i}^{xx_i}, v_{t x_i}^t)\), such as to build two edge disjoint directed paths from \( v_{s x_i}^s \) to \( v_{t x_i}^t \). The node \( v_s \) is connected to each of the nodes \( v_{s x_i}^s, x_i \in C \) with an edge \((v_s, v_{s x_i}^s)\). For each variable \( x_i \), we introduce a source delayed unit weight passenger travel intention \( \psi_{x_i} = (v_s, v_{s x_i}^s) \). These travel intentions exactly exhaust the allowed total passenger delay \( d \) of the decision problem. Hence, in order to be a yes-instance of delay management, no travel intention \( \psi_{x_i} \) can be dropped, and none of the travel intentions we introduce for the clauses can have a strictly positive arrival delay.

For each clause \( c_j \in C \) we introduce two nodes, \( v_{c_j}^s, v_{c_j}^t \) and a source punctual unit weight travel intention \( \psi_{c_j} = (v_{c_j}^s, v_{c_j}^t) \). For each positive variable \( x_i \) in the clause, we add the edges \((v_{c_j}^s, v_{\neg x_i}^{xx_i}), (v_{x_i}^{xx_i}, v_{c_j}^t), (v_{c_j}^s, v_{\neg x_i}^{xx_i}), (v_{\neg x_i}^{xx_i}, v_{c_j}^t)\), and for each negative variable \( \neg x_i \) the edges \((v_{c_j}^s, v_{x_i}^{xx_i}), (v_{x_i}^{xx_i}, v_{c_j}^t), (v_{c_j}^s, v_{x_i}^{xx_i}), (v_{x_i}^{xx_i}, v_{c_j}^t)\). Thus, the travel intention of each clause can be routed through any of the subgraphs constructed for the clause’s variables, and through the node \( v_{s x_i}^X \), where \( X \) is the negation of the literal as \( x_i \) appears in the clause \( c_j \).

The idea of the whole construction is as follows: if the variable
Figure 2.4.1: Reduction from 3-SAT to decision dynamic path choice delay management problem: we show the subgraph constructed for each variable $x_i$, as well as the construction for one literal of a generic clause $c_j = (x_1 \lor \neg x_2 \lor \neg x_3)$, and a sketch for $c_k = (x_3 \lor \neg x_4 \lor \neg x_5)$.

$x_i$ in the 3-SAT formula is set to true, the passenger travel intention $\psi_{x_i}$ is routed through the node $v^s_{x_i}$; if it is false, it is routed through $v^t_{x_i}$. The travel intention $\psi_{c_j}$ of each clause $c_j$ is routed through the subgraph of one of the variables satisfying the clause. If the formula is satisfiable, a routing inducing no additional delay exists for the constructed instance.

The reduction is linear in the problem size: we introduce $O(n + m)$ nodes, edges and travel intentions. We claim that the 3-SAT instance is a yes instance if and only if the constructed decision dynamic path choice delay management problem instance is a yes instance.

We remark that the decision dynamic path choice delay management problem is a yes instance if and only if it is possible to route the source delayed travel intentions through legs that wait and the source punctual travel intentions through legs departing as scheduled.

Assume the 3-SAT formula is satisfiable. We route the source delayed travel intentions through the node corresponding to the truth value the variable assumes. For each clause, we route the source punctual travel intention through the node corresponding to a variable satisfying the clause, which are routed through the negated value node of their respective variable. These routes cause no additional delay.
2.4. Restrictions, equivalences and a first reduction

Assume there is a delay policy. We assign the value corresponding to the node the source delayed travel intention \( \psi_{x_i} \) is routed through to each variable \( x_i \). By construction, each clause \( c_j \) is satisfied by the variable corresponding to the node the travel intention \( \psi_{c_j} \) is routed through.

The reduction also shows the inapproximability for the objective of minimizing the additional passenger delay. Indeed, we compare a zero-valued optimal solution with a non-zero approximate solution. The same holds for the total passenger delay objective: we can set an arbitrarily large period \( T \) and an arbitrarily high weight for the travel intention of each clause. Such choices, although somewhat unrealistic in practice, arbitrarily penalize any approximate solution dropping any travel intention \( \psi_{x_i} \) or delaying one travel intention \( \psi_{c_j} \).

Finally, the reduction also points out that with this setting, the component of disjointly routing source delayed and source punctual travel intentions alone makes this problem variant hard. Indeed, the initial question to be answered is if a routing exists which does not increase the delay of the network. This question alone already makes the problem hard. Only in a second stage do the decisions to drop some and delay some other travel intentions enter the game.
Chapter 3

Offline delay management

If you say that you are mine  
I’ll be here till the end of time  
So you got to let know  
Should I stay or should I go?  
*Should I stay or should I go? The Clash.*

This chapter focuses on the version of delay management where all information is known at the beginning of the optimization (the offline version). From a practical point of view, an offline approach corresponds to taking a snapshot of the network, and optimize the resulting instance without taking any other possible and maybe even related delays into account. In some sense, such an approach is blind to how things could evolve. An online version of delay management is discussed in Chapter 4. In this chapter, we mostly focus on the model of Section 2.3.2 where trains services have no intermediate stops and no slack time, that is, to the so-called binary delay management problem.

Our hardness results show that this model does indeed capture the crucial aspect of delay propagation of the delay management problem we consider. The polynomial-time algorithms, on the other hand, allow us to classify and draw a quite sharp boundary on the aspects which render the problem hard to solve. Two main parameters have an effect on the boundary between hard and easy problems. First, the topology of the railway network which is operated. If the net-
works have a simple topology, as for instance the topology of a path, the problem is easy to solve. As soon as the railway network topology becomes slightly more complex, as for instance on series-parallel graphs, the problem is in general hard. The latter result stresses the hardness of delay management, since it shows that the problem is hard on the simplest graphs with bounded treewidth, where other problems such as independent set admit a polynomial time algorithm if the graph is provided together with its tree-decomposition. Of bigger impact is the influence of the number of passenger transfers per passenger path. If source punctual passengers paths transfer at most twice, the problem admits a polynomial time solution by a reduction to a minimum cut problem. On the other hand, source punctual passenger paths transferring three times or more in general render the problem hard. Once we have assessed these results, we briefly show their implication on slightly more complex problem variants, such as for the binary delay management problem with intermediate stops.

In this chapter, we also address the problem variant which includes slack times. It turns out that slack times make the problem even more difficult to solve: the boundary between easy and hard instances lies by two (instead of three) transfers per passenger path. Restrictions in graph topology are also less effective, as the problem is hard to solve already on network topologies of a railway corridor, although with more than two transfers per passenger path.

**Summary of results**

Most of the results contained in this chapter are joint work with other persons. The hardness results are joint work with Leon Peeters, Riko Jacob and Anita Schöbel. The basics of the minimum-cut based approaches are joint work with Björn Glaus and Peter Widmayer. The remaining polynomial-time approaches are joint work with Leon Peeters, Riko Jacob and Peter Widmayer.

As a first step, we answer the question about the computational complexity of the decision version of the binary delay management problem. Then, we give two different approaches for efficiently solving restricted settings of the binary delay management problem: a minimum-cut based approach for a restricted number of transfers per passenger path and some other special cases, and a dynamic program for railway networks with the topology of a (graph-theoretic) path (a
3.1. Hardness of the binary delay management problem

Table 3.1: Classification of the binary delay management problem, with implied entries greyed out.

<table>
<thead>
<tr>
<th>Network topology</th>
<th>( \leq 2 ) transfers</th>
<th>( \leq 3 ) transfers</th>
<th>( &lt; \infty ) transfers</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>min cut</td>
<td>( \mathcal{NP} )-complete</td>
<td>( \mathcal{NP} )-complete</td>
</tr>
<tr>
<td>Series-parallel</td>
<td>min cut</td>
<td>( \mathcal{NP} )-complete</td>
<td>( \mathcal{NP} )-complete</td>
</tr>
<tr>
<td>Line</td>
<td>min cut</td>
<td>dyn. program</td>
<td>dyn. program</td>
</tr>
</tbody>
</table>

railway corridor).

Then, we show the implications of our hardness results and our minimum-cut approaches on the binary delay management problem with intermediate stops.

Next, we address the binary delay management problem with slack times and without intermediate stops and give proofs of hardness for the general problem variant, and show that several cases which are polynomial time solvable without slack times get hard by introducing slack times. Finally, we look at several restricted problem variants of the binary delay management problem with slack times and intermediate stops which can be solved efficiently by sensibly exploiting the slack times.

The classification of the binary delay management problem without and with slack times are summarized in Tables 3.1 and 3.2, respectively.

Unless otherwise stated, throughout this chapter we use the notation introduced in Section 2.3.2 for an instance of the different variants of the delay management problem. Thus, \( \mathcal{P} \) refers to the set of passenger paths of the considered instance, \( \mathcal{R} \) to the set of train services, \( G = (V, E) \) to the railway network, and so on.

3.1 Hardness results for the binary delay management problem

In this section, we consider the binary delay management problem. We first show that the decision version of the binary delay management problem is \( \mathcal{NP} \)-complete with passenger paths transferring at most three times and on a series-parallel network. Next, we show
Chapter 3. Offline delay management

Table 3.2: Classification of the binary delay management problem with slack times, with implied entries greyed out.

<table>
<thead>
<tr>
<th>Network topology</th>
<th>≤ 1 transfers</th>
<th>≤ 2 transfers</th>
<th>&lt; ∞ transfers</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>min cut</td>
<td>(\mathcal{NP})-complete</td>
<td>(\mathcal{NP})-complete</td>
</tr>
<tr>
<td>Series-parallel</td>
<td>min cut</td>
<td>(\mathcal{NP})-complete</td>
<td>(\mathcal{NP})-complete</td>
</tr>
<tr>
<td>Line</td>
<td>min cut</td>
<td>?</td>
<td>(\mathcal{NP})-complete</td>
</tr>
<tr>
<td>Tree, single dest.</td>
<td>“pedal-to-the-metal”</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

the implications stemming from the given reductions on several network parameters. Finally, we show that it is hard to approximate the additional passenger delay of the binary delay management problem.

3.1.1 Proof of hardness with three transfers

We now constructively give the proof of hardness for the binary delay management problem with passenger paths transferring at most three times, on a series-parallel network and for unit weight passenger paths. To that end, we start by giving the ideas of the construction and first show statements of hardness for settings more complex than the one just claimed, thus starting by showing weaker results.

**Definition:** Decision binary delay management problem.

**Instance:** A binary delay management instance, \(d \in \mathbb{N}\).

**Question:** Is there a delay policy for the given instance of the binary delay management problem, such that the total passenger delay is less than or equal to \(d\)?

**Theorem 3.1.** The decision binary delay management problem with passenger paths with at most four transfers is \(\mathcal{NP}\)-complete.

**Proof.** It is easy to see that the problem is in \(\mathcal{NP}\), as the weighted delay of the passenger paths induced by a delay policy \(\pi\) can be computed in polynomial time, and the size of \(\pi\) is polynomial as well. We show that the problem is \(\mathcal{NP}\)-hard by reduction from Maximum Independent Set [20, Problem GT20]. Let the undirected graph \(G_1 = (V_1, E_1), |V_1| = n, |E_1| = m\) be a Maximum Independent
3.1. Hardness of the binary delay management problem

Set instance asking for an independent set of size $K$. Consider its 2-subdivision $[51] G_2 = (V_2, E_2)$, i.e., the graph obtained by inserting the nodes $u_e, v_e$ for each undirected edge $e = \{u, v\} \in E_1$ and splitting the edge into three undirected edges $\{u, u_e\}, \{u_e, v_e\}, \{v_e, v\}$. We refer to this construction for an edge $e = \{u, v\}$ of the original graph $G_1$ as the extended edge in the 2-subdivision, symbolized by $(u, u_e, v_e, v)$. The graph $G_1$ has a maximum independent set of size $K$ if and only if its 2-subdivision $G_2$ has a maximum independent set of size $K + m$.

In the following, we construct gadgets for every extended edge of the 2-subdivision graph. In the resulting delay management instance, a specific set of maintained passenger paths in an optimal delay policy $\pi^*$ correspond to the nodes within the maximum independent set of the 2-subdivision.

For each node $q$ in $G_2$ we construct a passenger path $P_q$ in the delay management instance, such that two nodes $q, r \in V_1$ can be in the same independent set if and only if the corresponding passenger paths $P_q$ and $P_r$ can both be maintained in the same optimal delay policy. A maximum independent set in $G_2$ hence corresponds to an optimal set of maintained passenger paths.

For this construction, shown in Figure 3.1.1, consider an extended edge $(u, u_e, v_e, v)$. For the nodes $u, v$ we have passenger paths $P_u$ and $P_v \in P$, both source delayed and with unit weight. These passenger paths exist once for every node $u \in V$. Furthermore, we introduce passenger paths $P_{u_e}, P_{v_e}$ for $u_e$ and $v_e$, both with unit weight and source punctual. The exact configuration of all these passenger paths is shown in Figure 3.1.1. The legs and the train services (each serving one leg) are defined by means or the introduced passenger paths.

For each extended edge $(u, u_e, v_e, v)$ of the 2-subdivision we introduce five passenger paths in the binary delay management problem instance, $P^e_1, P^e_2, P^e_3, P^e_\alpha, P^e_\beta$, each with weight $w(P^e_i) = M, i \in \{1, 2, 3, \alpha, \beta\}$, where $M$ is a sufficiently large value, which we specify later. The passenger paths $P^e_\alpha$ and $P^e_\beta$ are source delayed, the other passenger paths are source punctual. Because of the large weight $M$, the source delayed passenger paths $P^e_\alpha, P^e_\beta$ are never dropped in an optimal delay policy $\pi^*$. For the same reason, the passenger paths $P^e_i, i \in \{1, 2, 3\}$ are always kept punctual. We refer to these passenger paths as $M$-paths, and Figure 3.1.1 shows their exact configura-
Let $\pi^*$ be an optimal delay policy for the constructed instance of the binary delay management problem, which implies that no $M$-paths are dropped. In $\pi^*$, the passenger paths corresponding to nodes of $G_2$ interact by sharing edges. Because of the $M$-paths, $\pi^*$ cannot maintain two interacting unit weight passenger paths, since one requires the shared leg to wait, whereas the other requires it to be on time in order not to be dropped. Indeed, $P_u$ and $P_{u_e}$ share the edge $(A_u, B_u)$, and $P_{u_e}$ and $P_{v_e}$ share $(D_e, E_e)$. Note that this construction enforces that each maintained unit weight passenger path has an arrival delay of $\delta$.

Hence, the unit weight passenger paths that are maintained in $\pi^*$ correspond to an independent set $I$ in $G_2$. Since every maintained passenger path reduces the cost of the delay policy, $I$ is a maximum independent set.

More precisely, set $\delta = 1$, $T = 2$, $M = m + 2$, and $d = 2mM\delta + (2m + n)T - (m + K)(T - \delta)$. Now $G_2$ has an independent set of size $m + K$ if and only if the binary delay management problem instance has a delay policy $\pi$ of cost at most $d$, i.e., which maintains $m + K$ unit weight passenger paths.

Finally, observe that the longest constructed passenger paths (of type $P_{v_e}$) require 4 transfers.
3.1. Hardness of the binary delay management problem

Theorem 3.1 can be strengthened by constructing an even simpler instance of the delay management problem. This reduction involves a much simpler network, namely a network which is series-parallel. We shortly discuss the main features of such graphs. Series-parallel graphs have treewidth two, which intuitively means they are almost trees. Many \( \mathcal{NP} \)-complete problems, such as Independent Set and Vertex Cover, become polynomial-time solvable on bounded-treewidth graphs. Hence, an \( \mathcal{NP} \)-hardness result for series-parallel graphs in some sense complements a polynomial time algorithm for trees. For a discussion of series-parallel graphs and treewidth, see for example [11]. The (directed) series-parallel graphs are defined recursively as follows. Every series-parallel graph has a designated source node \( s \) and a sink \( t \). The graph consisting of one edge from \( s \) to \( t \) is a series-parallel graph. Given two series-parallel graphs \( G \) and \( H \), their parallel composition and their serial composition are both series-parallel graphs. The parallel composition is obtained by identifying the two source nodes and the two sink nodes. The serial composition is defined by identifying the sink of \( G \) with the source of \( H \), and by defining the new source as the source of \( G \), and the new sink as the sink of \( H \).

Now, we give the following Theorem.

**Theorem 3.2.** The decision binary delay management problem on a series-parallel graph with passenger paths transferring at most three times and unit weight passenger paths is \( \mathcal{NP} \)-complete.

**Proof.** We modify the construction of Theorem 3.1 as follows. To reduce the maximal number of transfers to 3, the last edge \((F_e, G_e)\) is omitted for each \( e \in E_1 \). This construction still enforces that one of the passenger paths \( P_{v_e} \) and \( P_{u_e} \) must be dropped. With this construction, dropping \( P_{u_e} \) and maintaining \( P_{v_e} \) causes a delay of \( T \), which is less than the cost \( \delta + T \) of maintaining \( P_{u_e} \) and dropping \( P_{v_e} \).

Set \( \delta = 1, T = m + 2, M = m + 3, w(P_i) = 1, i \in \{u, v, u_e, v_e\} \) and \( d = 2n - K + mn - mK + 2m^2 + 6m \). We claim that \( G_2 \) has an independent set of size \( m + K \) if and only if the modified binary delay management instance has a delay policy of cost at most \( d \), which maintains \( K \) unit weight passenger paths of type \( P_u \) and \( m \) unit weight passenger paths of type \( P_{u_e} \). Such a policy contributes to the objective with weight \( K\delta \) for the maintained passenger paths \( P_u \), with \( (n - K)T \) for the dropped passenger paths of the same type,
with \( m\delta M \) for the source delayed \( M \)-paths, and with weight \( mT \) for the dropped passenger paths \( P_{u_e} \) and \( P_{v_e} \). Although dropping the passenger paths \( P_{u_e} \) and \( P_{v_e} \) contributes to the objective with weight \( T \) per passenger path, maintaining \( P_{u_e} \) has a different impact on the objective than maintaining \( P_{v_e} \). Indeed, maintaining \( P_{v_e} \) causes no addition to the objective, whereas maintaining \( P_{u_e} \) increases the objective by \( \delta \). Over all edges \( e \), this contribution can nevertheless be bounded by \( m\delta \). Adding all the above terms yields the value of \( d \).

Delaying or dropping \( M \)-paths is more expensive than dropping unit weight passenger paths. Further, dropping more than \( n - K + m \) unit weight passenger paths causes the delay to exceed \( d \), as can be verified. Thus, the construction enforces at least \( K + m \) passenger paths to be maintained, that is, at most \( n - K + m \) passenger paths to be dropped.

To make the underlying network series-parallel, we identify all nodes of one position into one single node, see Figure 3.1.2. That is, all nodes \( A_u \) are contracted into one node \( A \), all nodes \( B_u \) into one node \( B \), all \( C_e \) into \( C \), and so on. Then, the graph consists of the 6 nodes \( A, B, C, D, E, F \), with bundles of parallel edges between \( (A, B) \), \( (B, C) \), \( (B, D) \), \( (C, D) \), \( (D, E) \), and \( (E, F) \).

In the resulting series-parallel network, each passenger path still uses the same edges as prior to contracting the nodes. For example, the passenger path \( P^* \) in Figure 3.1.2 uses the edges \( f_1, f_2, f_3, f_4 \) in the series-parallel network. The interaction of the gadgets in the series-parallel network is achieved by passenger paths that share an edge.

Observe that the weight \( M = m + 3 \) is polynomial in the size of the instance. Thus, as a last step, we can transform the reduction to an instance with unit weight passenger paths by introducing \( M \) parallel passenger paths of unit weight for each \( M \)-path.

Independent set is known to be hard for graphs with bounded degree \( \Delta \geq 3 \). The strongest reduction to a series-parallel network naturally creates a graph with very few nodes, each with very high degree. However, if we give up series-parallelness of the delay management network, the bounded degree hardness of Independent set implies, with the reduction above, that delay management is hard on a network with in-degree bounded by two (attained by the node of type \( D_e \) in Figure 3.1.2) and out-degree three.
3.1. Hardness of the binary delay management problem

Figure 3.1.2: At the top, a network stemming from Independent Set on nodes \( u, v, w \) and edges \( e = \{u, v\}, e' = \{v, w\} \). Except for \( P^* \), the passenger paths of the construction are omitted. Capital letters are the node labels. At the bottom, the corresponding series-parallel network after the contraction has taken place. Note that the passenger path \( P^* \) is routed along the edges \( f_1, f_2, f_3, f_4 \), as in the uncontracted version.

Finally, we remark that delay management is not fixed parameter tractable in the number of transfers per passenger path, as it is hard already for a constant number of transfers per passenger path.

Approximating the delay

The reduction above additionally provides a weak inapproximability result for the general delay management problem. Indeed, we can show that the objective value of every feasible policy of the instance resulting from the above reductions are polynomially bounded in the size of the input. Given that \( P \neq \mathcal{NP} \), for problem where this holds for any problem instance, this fact disproves the existence of an FPTAS, since the corresponding decision problem could otherwise be solved by choosing an appropriate \((1 + \varepsilon)\)-approximation in time polynomial
in the instance size and in $\frac{1}{\varepsilon}$ [6].

In our case, the existence of an FPTAS would allow us to solve the decision variant of Independent Set in polynomial time. Indeed, a good enough approximation of delay management would allow to deduce the existence of an independent set of a specific cardinality.

**Corollary 3.3.** The binary delay management problem does not admit a FPTAS, unless $P \neq NP$.

**Proof.** Assume the contrary were true, and let $A$ be the algorithm which approximates the objective value of delay management within a factor $(1 + \varepsilon)$ in time polynomial to the input size and $\frac{1}{\varepsilon}$. Consider the instance of the decision binary delay management problem stemming from the reduction of Theorem 3.2, and choose $\varepsilon < \frac{1}{d} = (2n - K + mn - mK + 2m^2 + 6m)^{-1}$. In this case, $A$ gives an approximate solution of objective value at most $(1 + \varepsilon)d$. Therefore, if a solution of cost $d$ exists, $A$ returns a solution of cost strictly smaller than $(1 + \frac{1}{d})d = d + 1$. Since the inequality is strict and only integral values are admissible for feasible solutions, the algorithm returns a solution of cost $d$ if any such solution exists, and does so polynomial in the input size and $\frac{1}{\varepsilon} = 2n - K + mn - mK + 2m^2 + 6m = O(m^2)$, thus allowing the solution of an $NP$-complete problem.

### 3.1.2 Single delayed leg

The construction above can be used also to show the hardness of the setting described in Section 2.4.2 with one initially delayed leg, with the exact same number of maximal transfers, unit weight passenger paths and a series-parallel network

**Theorem 3.4.** Consider the decision version of the delay management problem stemming from the binary delay management problem with the difference that the source delays are specified by one single delayed leg and not through passenger paths. This problem, on a series-parallel network with passenger paths transferring at most three times and unit weight passenger paths, is $NP$-complete.

**Proof.** Having proved Theorem 3.2, we only need to show that it is possible to introduce the source delayed passenger paths through a single delayed leg. Moreover, to do so, we may create passenger
paths with at most three transfers, which do not influence the previous construction harmfully and must ensure that the network remains series-parallel. To that aim, we extend the series-parallel network of Figure 3.1.2 as follows, and shown in Figure 3.1.3. The main idea is to introduce new legs which interact exclusively with source delayed passenger paths, and extend these passenger paths by first making them board the delayed edge and bring them to their original source with an additional leg. To that aim, we first introduce the delayed edge $e_0$, with head in node $A$. Such edge can be added to the existing network with a serial composition. Then, we introduce a parallel edge $e_1$ between the nodes $A$ and $B$, and a second parallel edge $e_2$ between the nodes $B$ and $C$. These edges can be added with a parallel composition when adding the complete bundle of edges between the respective nodes. Thus, the resulting network is series-parallel. Now, we remain with routing the source delayed passenger paths from $e_0$ to their respective source. All source delayed passenger paths with source in $A$ (and thus traveling to $B$ in our original construction) need only transfer in $A$ from $e_0$ to their respective edge. Hence, we induce a single transfer for each of these passenger paths by adding $e_0$ as a first leg of these passenger paths. For the source delayed $M$-paths from $C$ to $D$, we make them transfer first from $e_0$ to $e_1$, then from $e_1$ to $e_2$ and finally form $e_2$ to their respective edges. Thus, we add the consecutive legs $e_1$ and $e_2$ as first two legs of these passenger paths. In this way, we have introduced three transfers per passenger path. It is clear that the legs $e_1$ and $e_2$ wait for the delayed passenger paths, as they are exclusively boarded by source delayed passenger paths. Moreover, due to their high weight, $M$-paths are maintained in every optimal solution. Finally, since the passenger paths with source delay reach their original source using only newly introduced edges, the behavior of the construction remains unchanged.

3.1.3 Dynamic passenger path choices

We established hardness for the problem variant with dynamic path choices, the decision dynamic path choice delay management problem, in Theorem 2.2. However, that problem variant did not restrict the network topology to a series-parallel network, which we would like to achieve in this section.
Figure 3.1.3: *The extended construction of Theorem 3.4.* Dotted edges are the legs of the previous construction, whereas the solid edges are introduced with this construction.

Unfortunately, dynamic path choices render the construction in the proof of Theorem 3.2 useless. To see this, note that all passenger paths with the same “function” in the construction of Theorem 3.1 start and end at the same nodes in the series-parallel network. For example, all passenger paths $P_v, v \in V_1$, start at $A$ and end at $B$, and all passenger paths $P^e_v, e = \{u, v\} \in E_1, v \in V_1$, start at $A$ and end at $F$. Consider the following delay policy. For each bundle of parallel edges, we make one leg depart on time, make one leg wait, and set all other legs arbitrarily. Now, each source punctual passenger path can dynamically choose the punctual edges, and each source delayed passenger path can dynamically choose the waiting edges. Clearly, this solution does not drop any passenger paths, nor does it delay any source punctual passenger paths.

As a remedy, the construction can be modified such that the route of each passenger path is unique in the series-parallel network. The construction uses a 4-subdivision instead of a 2-subdivision. To this aim, each of the $2m$ parallel subgraphs between $B$ and $D$ is replaced by a serial interaction gadget, with one source punctual $M$-path and one source delayed $M$-path. Further, the passenger paths between $B$ and $D$ in the original construction are split, see Figure 3.1.4. In this modified construction, only the nodes $B_u$ and $D_e$ are contracted. This construction can be embedded in a series-parallel graph. The resulting graph before the contraction is depicted in Figure 3.1.4. As a result, after contracting the nodes $B_u$ and $D_e$, each passenger path has a unique feasible route.

**Corollary 3.5.** *The decision dynamic path choice delay management problem on a series-parallel graph with passenger paths transferring at most three times and unit weight passenger paths is NP-complete.*
3.1. Hardness of the binary delay management problem

3.1.4 Approximating the additional passenger delay

As stated in Section 2.2, our objective is to minimize the total passenger delay. Alternatively, it also makes sense to minimize additional passenger delay (as defined in Section 2.3.1), that is, the (weighted) delay that passenger paths face in addition to their source delay. Indeed, as there are no slack times, a source delayed passenger path can never do better than arrive at its destination with a delay of $\delta$. This portion of the delay cannot be optimized, so it is reasonable to omit it from the objective function.

As Independent Set and Vertex Cover are complementary problems, the results from [29] provide an inapproximability result for the delay management problem with the additional passenger delay objective. The proof involves a different reduction from independent set, and generally needs more than three passenger transfers.

Lemma 3.6. For any $\epsilon > 0$, it is $\mathcal{NP}$-hard to approximate the binary decision delay management problem with the objective of minimizing the additional passenger delay within a factor $\frac{15}{14} - \epsilon$.

Proof. It is easy to see that the problem is in $\mathcal{NP}$, as the weighted arrival delay of the passenger paths induced by a delay policy can be computed in polynomial time, and the size of the delay policy is polynomial as well. We first construct an alternative reduction from Independent Set [20, Problem GT20] to show that the problem is $\mathcal{NP}$-hard, and then use this reduction to prove the inapproximability result.

The idea of the alternative reduction is the following. For each...
Chapter 3. Offline delay management

Figure 3.1.5: An example of the reduction. At the left, the independent set graph, at the right, the resulting delay management network. Passenger Paths are drawn as arrows to indicate their direction.

node in the independent set instance, we insert a so-called vertex-path, a passenger path that is maintained if and only if the node is in the independent set. As in the proof of Theorem 3.1, two adjacent nodes yield two vertex-paths that share a leg in such a way that at most one of the two passenger paths can be maintained in any optimal delay policy. There is a one-to-one correspondence between the nodes of an independent set and the maintained passenger paths in the resulting binary delay management problem instance.

More precisely, given an instance $G_I = (V_I, E_I), K \in \mathbb{N}, |V_I| = n, |E_I| = m,$ of Independent Set, we construct an instance of the binary delay management problem as follows, sketched in Figure 3.1.5, using the notation introduced in the problem definition. Set $\delta = 1, T = 2, d = 2n - K,$ and consider an arbitrary ordering of $V_I$ from 1 to $n$ and identify the nodes of $V_I$ with this numbering. For presentation purposes, the construction is embedded in the plane, with ‘time’ on the horizontal axis, and ‘space’ on the vertical axis. For each node $i \in V_I,$ we construct a unit weight source punctual vertex-path $P_i.$ We specify the edges of the passenger path below. The passenger path $P_i$ starts at station $i$ at time zero, reaches station 0 at time $i,$ and ends its travel intention at station $n - i$ at scheduled time $n$ (so, $t_{d(P_i)} = n$). For this travel intention, the passenger path $P_i$ needs to connect between legs several times.

At station 0 we introduce a so-called corner gadget, illustrated in...
3.1. Hardness of the binary delay management problem

3.1.1. Hardness of the binary delay management problem

Figure 3.1.6: The corner gadget, enforcing a vertex-path to be dropped if already delayed before station 0, and delaying it after 0. Thick passenger paths are $M$-paths, dashed passenger paths have source delay.

Figure 3.1.7: The crossing gadget, enforcing two passenger paths whose nodes share an edge in $G_I$ to use a common leg in the delay management problem. Here, passenger paths are drawn as arrows to indicate their direction.

Figure 3.1.6. Each passenger path $P_i$ arrives at its node for station 0 with an inbound edge it shares exclusively with a source punctual passenger path $P_{0i}$ of weight $M$. The interaction of $P_{0i}$ with the network is limited to this edge, and to the passenger path $P_i$. Similarly, $P_i$ leaves the node corresponding to its station 0 through an outbound edge shared exclusively with a source delayed passenger path $P_{i\delta}$ of weight $M$, which interacts with the network solely through this edge.

By choosing $M$ big enough, delaying $P_{0i}$ is more expensive than dropping $P_i$. In this way, $P_i$ is dropped if it reaches the edge in common with $P_{0i}$ with a delay. Similarly, $P_i$ is delayed if it reaches the edge common with $P_{i\delta}$ on time. We refer to the passenger paths of weight $M$ as $M$-paths.

Two adjacent nodes $i, j \in V_I, i < j$, cannot both be in the independent set. To enforce this, the passenger paths $P_i$ and $P_j$ cross exactly once on one common edge, in a so-called crossing gadget as shown in Figure 3.1.7. In the embedding, the crossing gadget is placed after $P_i$’s corner gadget but before $P_j$’s corner gadget, see Figure 3.1.5. Thus, given the decision to maintain passenger path $P_i$, the passenger path $P_j$ must be dropped, as it reaches its corner gadget with a delay. Conversely, given the decision to maintain the passenger path $P_j$, the passenger path $P_i$ must be dropped, as it reaches the crossing gadget with a delay. Hence, the two passenger paths cannot be maintained concurrently if there is an edge $\{i, j\} \in E_I$. Note that
the passenger paths are disjoint if \( \{i, j\} \not\in E_I \), and simply “cross” at a node (non-marked intersections of passenger paths in Figure 3.1.5).

Set \( M = 2n \). Delaying one of the \( M \)-paths causes a delay of \( 2n \cdot \delta = 2n \), which is equal to the delay caused by dropping all unit weight vertex-paths. However, at least one vertex-path can always be maintained by neither delaying nor dropping any \( M \)-paths. In the worst case, all remaining vertex-paths must be dropped, which in total yields a weighted delay of \( 2n - 1 \) for the vertex-paths. Thus, delaying or dropping \( M \)-paths cannot lead to an optimal policy. Again, the size of the instance remains polynomial when each \( M \)-path is replaced by \( M \) unit weight passenger paths, as \( M = 2n \).

The graph \( G_I \) has an independent set of size at least \( K \) if and only if the constructed unit weight delay management instance has a delay policy inducing at most \( 2n - K \) additional passenger delay (which corresponds to a total passenger delay of at most \( 2n - K + nM\delta \)). The reduction is polynomial, since each node induces \( 1 + 2M = 1 + 4n \) passenger paths and \( O(m) \) edges, and the instance can be built in polynomial time.

Finally, we show the inapproximability result for the additional passenger delay objective. As the nodes of the maintained vertex-paths form an independent set in \( G_I \), the nodes of the dropped vertex-paths form a vertex cover in \( G_I \). Hence, \( G_I \) has a vertex cover of size at most \( c \) if and only if there exists a delay policy with additional passenger delay at most \( n + c \) (i.e, \( c \) source punctual vertex-paths are dropped). As shown in [29], it is \( NP \)-hard to distinguish graphs having a vertex cover of size \( \leq (\frac{6}{8} + \epsilon)n \) from those having a vertex cover of size \( \geq (\frac{7}{8} - \epsilon)n \). This provides an inapproximability result of \( \frac{7}{6} - \epsilon \).

Our additional passenger delay objective has an additive offset \( n \), since each vertex-path (which are all source punctual) have an arrival delay at least \( \delta = 1 \). Hence, distinguishing the above delay management instances with additional passenger delay \( n + \frac{6}{8}n = \frac{14}{8}n \) from those with additional passenger delay \( n + \frac{7}{8}n = \frac{15}{8}n \), is equivalent to distinguishing between the corresponding sizes of a vertex cover. The ratio between these values proves the statement. \( \square \)

In this case, the reduction proves the hardness of delay management on planar graphs with maximum in-degree two and out-degree two, and maximum degree 4 for each node. Note that the maximum
3.2 Minimum-cut based approaches

Having established that the general version of delay management is \(\mathcal{NP}\)-hard, we now focus on restricted problem variants which are solvable in polynomial time. In this section, we address the solution methods based on reductions to the minimum-cut problem. Unless otherwise stated, throughout this section we use the notation introduced in Section 2.3.2 for an instance of the binary delay management problem.

**Minimum cuts** We consider a weighted graph \(H = (U, F)\), where \(U\) is the set of nodes of the graph and \(F\) the set of edges. Let \(s, t \in U\) be two nodes of the graph, the source and the target node, respectively. Finally, let \(c : F \rightarrow \mathbb{N}\) be a function assigning nonnegative integral weights to the edges.

An \(s-t\)-cut \([S, \bar{S} = U \setminus S]\) of a weighted graph \(H = (U, F)\) as defined above is a partition of the node set \(U\) into two disjoint sets \(S\) and \(\bar{S}\). The partition is such that \(s \in S\) and \(t \in \bar{S}\). An edge \(e = \{u, v\}\) is said to traverse the cut if its endpoints are in different partitions; hence, if \(e\) traverses the cut, either \(u \in S\) and \(v \in \bar{S}\), or \(u \in \bar{S}\) and \(v \in S\). Given a cut, let \(E_c = \{e = \{v_i, v_j\} \in F : |e \cap S| = 1\}\) be the edges which traverse the cut. The weight of a cut is defined as the sum of the weight of the edges which traverse the cut: \(\sum_{e \in E_c} c(e)\). A \(s-t\)-cut is minimum if it minimizes, over all possible \(s-t\)-cuts, the weight of the edges traversing the cut.

A minimum cost directed \(s-t\)-cut \([S, \bar{S} = V \setminus S]\) of a weighted directed graph \(N = (U, F), c : F \rightarrow \mathbb{N}\) is a slight variant of the minimum cost cut problem. First, we consider directed graphs. Hence each edge \(e = (u, v) \in F\) is directed from \(u\) to \(v\). The node partition is again such that \(s \in S\) and \(t \in \bar{S}\). An edge \(e = (u, v)\) traverses the directed \(s-t\)-cut if \(u \in S\) and \(v \in \bar{S}\). Thus, the edges traversing the cut in a directed cut are specified as \(E_c = \{e = (v_i, v_j) \in F : v_i \in S, v_j \in \bar{S}\}\), and the weight of the directed \(s-t\)-cut is defined
as \( C = \sum_{e \in E_c} c(e) \). A minimum cost directed \( s-t \)-cut is such that it minimizes this weight.

Both problems are polynomial-time solvable for positive edge weight [16, 2], whereas they are \( \mathcal{NP} \)-complete if edges are allowed to have negative weight, as we can model max-cut by setting all edge weights to \(-1\) and enumerating all possible \( s, t \) pairs.

**Delay management as a minimum cut problem**  Given a binary delay management problem instance, our goal is to find a delay policy specifying which of the legs should wait. Hence, we need to partition the legs \( E \) into two sets. Let \( \Delta \subseteq E \) be the set of legs that wait for connecting passenger paths, and \( \Omega = E \setminus \Delta \) the set of legs that depart as scheduled.

The main idea of the cut-based approaches is as follows. Given an instance of the binary delay management problem, we build a new graph \( H \) such that any minimum weight partition \([S, \bar{S}]\) of the node set can be interpreted as a delay policy, and the weight of the cut corresponds to the cost of the delay policy. To that aim, some nodes in \( H \) represent the legs in \( E \). Now, among the nodes in \( H \) which represent the legs, we interpret the ones in the partition \( S \) as legs that wait for connecting passenger paths, and the ones in \( \bar{S} \) as legs that depart as scheduled.

To account for the correct cost of the delay policy within the cut, we introduce directed weighted edges in the cut instance, some of them parallel. In order not to blow up the graph artificially and to be able to use algorithms which assume the graph to be simple, these parallel edges can be contracted to a single edge having the combined weight of the parallel edges it substitutes. In the proofs to follow, for simplicity we analyze the non-simple graph.

In the remainder of this section, we first show the transformation for passenger paths transferring at most twice. Then, we extend the transformation to a slightly more general case, and finally apply the cut-based approach to restricted delay policies and to restricted passenger path settings.
3.2.1 Passenger paths with at most two transfers

Given a binary delay management problem instance $Q$, with the restriction that each passenger path transfers at most twice, we build an equivalent weighted graph $H = (U, F)$, $s, t \in U$, $c : F \to \mathbb{N}$ as an instance $I(Q)$ of the minimum directed $s$-$t$-cut problem as follows. We point out that the transformation given in the following can be simplified for short passenger paths. These simplifications are addressed later in this section.

We set the node set to $U = \{s, t\} \cup \{v_e : e \in E\} \cup \{v_p : p \in P\}$. Hence, there are new source and target vertices $s$ and $t$, a node $v_e$ for each leg $e \in E$ and a node $v_p$ for each passenger path $P \in P$. The edge set $F$ and the weights of the edges are built on the basis of the existing passenger paths $P$.

For each source delayed passenger path $P = \{e_1, \ldots, e_k\} \in P : D(P) = \delta$ built by $k \leq 3$ legs we introduce the following construction, illustrated in Figure 3.2.1 for $k = 3$. First, we introduce an edge $(s, t)$ with weight $c(s, t) = \delta w(P)$ into $F$. This edge accounts for the source delay of the passenger path $P$ and must always be accounted for in any delay policy, as it cannot be optimized. For each leg in the passenger path, we introduce the directed edge $(v_p, v_{e_i})$, $\forall e_i \in P$ with weight $c(v_p, v_{e_i}) = +\infty$. Finally, we introduce an edge $(s, v_P)$ with weight $c(s, v_P) = (T - \delta) w(P)$. The idea of the construction is to force $v_P$ into $\bar{S}$ as soon as at least one of the vertices $v_{e_i}, e_i \in P$ is in $\bar{S}$. This behavior is achieved using the infinite-weight edges. In a directed cut of non-infinite weight such edges do never traverse the cut from $S$ to $\bar{S}$. From a delay management point of view, $v_P$ represents the state of $P$: the passenger path is maintained if $v_P \in S$ (since all its legs wait), whereas if at least one of the legs departs as scheduled $v_P \in S$, thus dropping the passenger path. If $v_P \in \bar{S}$, the edge $(s, v_P)$ contributes an additional $(T - \delta) w(P)$ to the weight of the passenger paths, thus correctly accounting for the delay caused by dropping $P$.

For each source punctual passenger path $P = \{e_1, \cdots, e_k\} \in P : D(P) = 0$ built by $k \leq 3$ legs we use the following construction, illustrated in Figure 3.2.2 for $k = 3$. First, we introduce the edges $(v_{e_i}, v_P), e_i \in P$ of weight $c(v_{e_i}, v_P) = +\infty$ into $F$. Next, we introduce the edge $(v_P, t)$ with weight $c(v_P, t) = \delta w(P)$. Finally, we introduce the edges $(v_{e_i}, v_{e_{i+1}}), i \in \{1, \cdots, k - 1\}$ of weight
Chapter 3. Offline delay management

Figure 3.2.1: The construction for a source delayed passenger path $P = \{e_1, e_2, e_3\}$.

Figure 3.2.2: The construction for a source punctual passenger path $P = \{e_1, e_2, e_3\}$.

c\left(v_{e_i}, v_{e_{i+1}}\right) = (T - \delta)w_P$. In a similar way to the previous construction, the infinite-weight edges ensure that $v_P$ is in $S$ as soon as at least one of the vertices $v_{e_i}$ is in $S$. The edge between $v_P$ and $t$ accounts for the delay $\delta$ in case the passenger path is delayed, and the edges $(v_{e_i}, v_{e_{i+1}})$ between the leg nodes $v_{e_i}$ account for the increase in delay caused by dropping the passenger path $P$.

Thus, for every instance $Q$ of the binary delay management problem with the restriction that passenger paths transfer at most twice we can build a corresponding instance $I(Q)$ of the minimum directed $s$-$t$-cut problem.

**Lemma 3.7.** Consider an instance $Q$ of the binary delay management problem with passenger paths transferring at most twice, and an instance $I(Q)$ of the minimum directed $s$-$t$-cut problem on the weighted directed graph $H = (U, F)$ derived from $Q$. The cost of every minimum directed $s$-$t$-cut $[S, \bar{S}]$ in $H$ is equivalent to the
total passenger delay of the binary delay management problem instance \( Q \) when applying the delay policy \( \Delta = S^-, \Omega = \bar{S}^- \), where \( S^- = S \setminus \{ \{ s \} \cup \{ v_P : P \in \mathcal{P} \} \} \), \( \bar{S}^- = \bar{S} \setminus \{ \{ t \} \cup \{ v_P : P \in \mathcal{P} \} \} \).

Proof. Since no infinite-weight edge leave \( s \), the partition \( S = \{ s \}, \bar{S} = V \setminus \{ s \} \) proves the existence of a non-infinite weight directed \( s-t \)-cut. Hence, it is now sufficient to prove that each passenger path is penalized correctly for all waiting policies specified in the Theorem. It then follows that the complete solution \((\Delta, \Omega)\) has the correct weight, and that the cost of the cut corresponds to the total passenger delay for the binary delay management instance \( Q \). Because of the bijection between them, the cost of the minimum cut corresponds to the minimum cost delay policy for the binary delay management problem.

By construction of \( H \), the edges influenced by each passenger path are limited. Consider a source delayed passenger path \( P = \{ e_1, \cdots, e_k \} \). If \( v_{e_i} \in S, e_i \in P \), then \( v_P \in S \). Were this not the case, the edge \(( s, v_P )\) would traverse the cut, increasing the cut’s size unnecessarily. Hence, only the edge \(( s, t )\) traverses the cut, correctly accounting for the passenger path’s source delay, since in this situation the passenger path \( P \) is maintained. If \( \exists e^* \in P : v_{e^*} \in \bar{S} \) (at least one of the legs \( P \) wishes to board departs as scheduled), \( v_P \in \bar{S} \) follows because of the infinite-weight edge \(( v_P, e_i )\). Thus, the edge \(( s, v_P )\) traverses the cut, bringing the passenger path’s contribution to the cut’s size to \( Tw(P) \), correctly representing the cost of dropping \( P \). Note that for a leg \( e_j \in P \) which waits, the infinite-weight edge \(( v_P, v_{e_j} )\) traverses the cut from \( \bar{S} \) to \( S \), thus not increasing the cut’s weight, as we are considering a directed \( s-t \)-cut. Thus, source delayed passenger paths are correctly accounted for in this construction. Consider a source punctual passenger path \( P = \{ e_1, \cdots, e_k \} \). First, observe that due to the infinite weight edges \(( v_{e_i}, v_P )\), \( e_i \in P \), the node \( v_P \) is in \( S \) whenever at least one of the vertices \( \{ e_1, \cdots, e_k \} \) is in \( S \). If \( v_P \in \bar{S} \), it follows that \( \{ e_1, \cdots, e_k \} \subseteq \bar{S} \), and thus none of the edges related to \( P \) traverses the cut. When \( v_P \in S \), the edge \(( v_P, t )\) traverses the cut, contributing \( \delta w(P) \) to the cut’s weight, reflecting the weighted arrival delay for delaying passenger path \( P \). Finally, \( P \) can only be dropped when connecting from \( e_i \) to \( e_{i+1}, i \in \{ 1, \cdots, k - 1 \} \). For a drop to happen, we must have \( v_{e_i} \in S \) and \( v_{e_{i+1}} \in \bar{S} \), which increases the weight of the cut by \(( T - \delta)w(P) \), thus overall correctly accounting for the cost of drop-
ping a source punctual passenger path. It remains to be shown that at most one of these edges can traverse the cut. This directly follows from the fact that $k \leq 3$, thus there are at most two such consecutive edges for any passenger path $P$, and at most one of them can traverse the cut.

It follows that delay management with at most two transfers per passenger path can be solved efficiently, which leads to the following theorem.

**Theorem 3.8.** The minimum cost delay policy for the binary delay management problem where passenger paths transfer at most twice can be found efficiently by reduction to a polynomial size minimum cost directed s-t-cut problem.

**Proof.** When building the graph $H$, we introduce a node for each leg and for each passenger path of length three, i.e. $O(|E| + |P|)$ vertices. Similarly, we introduce one edge $(e, t)$ for each leg and $O(1)$ edges for each passenger path, i.e. $O(|E| + |P|)$ edges in total. Given a weighted directed graph with $n$ vertices and $m$ edges, a minimum directed $s$-$t$-cut can be computed in $O(mn \log n)$ time \([60]\). Hence, we need $O((|E| + |P|)^2 \log(|E| + |P|))$ time to find a delay policy minimizing the total passenger delay.

As already hinted earlier, the construction can be skimmed if passenger paths use fewer than three consecutive edges. In particular, slight changes in the construction render the use of the nodes $v_P, P \in P$ unnecessary for passenger paths involving fewer than two transfers. The skimmed constructions are given in \([21]\) and \([22]\) for the model variant with one source delayed leg, which can easily be adapted to the model used above. We stick to the construction above since it allows the following generalization.

**Corollary 3.9.** The minimum delay policy for the binary delay management problem where the number of transfers of source punctual passenger paths is no more than two can be found efficiently by reduction to a polynomial size minimum cost directed s-t-cut problem.

Note that in this theorem we do not constrain the number of transfers of source delayed passenger paths.
3.2. Minimum-cut based approaches

Proof. For the source punctual passenger paths we apply the same construction as for Lemma 3.7. For source delayed passenger paths, we apply the construction of Lemma 3.7 also for passenger paths with more transfers ($k > 3$). Note that with this extension, the condition that as soon as one leg departs as scheduled the corresponding passenger path-node $v_P$ is in $S$ still holds, thus still accounting correctly for the delay of dropping the passenger path.

Note that it is sufficient to set the weight of infinite-weight edges in the constructed graph $H$ to $1 + T \sum_{P \in \mathcal{P} : \Delta(P) = \delta} w(P)$, as it will then be more advantageous to drop all source delayed passenger paths rather than letting one of such edges traverse the cut.

Although the hardness results at the beginning of this chapter show that the problem is hard for three transfers per passenger path, one could be tempted to handle the source punctual passenger paths by merging the above two constructions for source delayed and source punctual passenger paths as follows. For each source punctual passenger path $P \in \mathcal{P}$, one could introduce two nodes, $v'_P$ and $v''_P$. Applying the construction for source punctual passenger paths on $v'_P$ we could force this node to be in $S$ as soon as one of the passenger path’s nodes is in $S$ (i.e, one of its legs waits for delayed passenger paths). Similarly, by applying the construction of source delayed passenger paths on $v''_P$ we could force this node to be in $\bar{S}$ as soon as one of the passenger path’s nodes is in $\bar{S}$. An appropriately weighted edge between these two nodes could then account for the delay of dropping the passenger path. Unfortunately, for a given delay policy, this construction cannot distinguish between a passenger path first boarding legs that depart as scheduled and eventually being delayed (but not dropped) from a passenger path first boarding a legs that wait and being dropped at some point afterwards. Any approach trying to incorporate the position of the drop by similar gadgets fails, since it is not possible to both guarantee that at most one of the edges accounting for the additional costs of dropping a passenger path traverses the cut and at the same time distinguish between a policy which delays and one that drops a passenger path. In some sense, the built structures allows us to insert only very simple implications into the minimum-cut problem.
3.2.2 Special cases

We extend the min-cut approach to several restricted settings of the binary delay management problem. First, we analyze a restricted network and passenger path topology, then we consider a restricted delay policy.

Out-Trees

We adapt the proposed reduction for the case where the delay management network $G$ is an out-tree, i.e., a directed tree where all edges are directed from the root node to the tree’s leaves, and all source delayed passenger paths depart from the root node. This setting has a nice interpretation: on the given network, this passenger path configuration is obtained if one leg in the network runs off-schedule, and as a consequence all passenger paths using that leg now have a (source) delay. A snapshot of this situation results in the given setting. The “never-meet-property” of [55] applies to out trees. Hence, the constraint matrix of the ILP-formulation proposed there for delay management is totally unimodular on out-trees, and the ILP can thus be solved efficiently by linear programming on these instances. Our reduction is another method to solve this special network structure by a purely combinatorial algorithm.

We transform the instance of the binary delay management problem into an instance of the minimum cost directed $s$-$t$-cut problem on a weighted directed graph $H = (U, F)$ with source and target nodes $s, t \in U$ and the weight function for the directed edges $c : F \to \mathbb{N}$. As before, all legs $E$ are mapped to nodes $U$; we add two nodes $s$ and $t$ to $U$. For each passenger path $P = \{e_1, \ldots, e_l\}, e_i \in E$, we add the following edges. An edge $(v_{e_1}, t)$ with weight $c(v_{e_1}, t) = \delta \cdot w(P)$, which accounts for the weighted arrival delay of a passenger path if it is delayed; $(v_{e_i}, v_{e_{i+1}}), \forall i \in \{1, \ldots, l - 1\}$ with weight $c(v_{e_i}, v_{e_{i+1}}) = (T - \delta) \cdot w(P)$, which accounts for the weighted arrival delay of a passenger path if, once delayed, it is dropped. Furthermore, for each source delayed passenger path $P = \{e_1, \ldots, e_l\}, e_i \in E$ we add an edge $(s, v_{e_1})$ with weight $c(s, v_{e_1}) = T \cdot w(P)$, accounting for the weighted arrival delay of the source delayed passenger path if it is dropped by its first leg.

Lemma 3.10. Consider an instance $Q$ of the binary delay manage-
ment problem where the railway network is an out-tree and all source delayed passenger paths have the tree’s root node as origin node. The cost of the minimum cost directed $s - t$-cut $[S, \bar{S}]$ on the graph $H$ constructed from $Q$ is equal to the total passenger delay of the minimum cost delay policy for $Q$.

One optimal policy is $\Delta = S \setminus \{s\}, \Omega = \bar{S} \setminus \{t\}$.

Proof. On out trees, an optimal delay policy for the binary delay management problem has the following structure: a leg $f$ may wait only if its feeder leg $e$ did also wait. Were this not the case, the leg $f$ would be waiting in spite of the fact that no delayed passenger paths have reached it. Indeed, all delayed passenger paths using $f$ are dropped by the scheduled departure of $e$. The same structure applies for the directed cut: for any node $v$ in $\bar{S}$, all nodes which can be reached from $v$ through the directed edges are also in $\bar{S}$. There are in fact no edges connecting nodes in $S$ to the reachable nodes, whereas there are some connecting them to $t \in \bar{S}$ (which would increase the costs of the cut were they in $S$).

Given a passenger path $P = \{e_1, e_2, \ldots, e_l\}$, let $j$ be the index of the last node $v_{e_j} \in S$ for $e_j \in P$ and $j = 0$ if no such node exists. We distinguish two cases. First, if $j = 0$, then none of the $P$’s legs wait for delayed passenger paths. Accordingly, for source delayed passenger paths the edge $(s, v_{e_1})$ traverses the cut, its weight effectively accounting for the arrival delay of dropping this passenger path. If $P$ is source punctual, no edges related to $P$ are in the cut, since all nodes reachable from $v_{e_1}$ are in $\bar{S}$. For the second case, $j \geq 1$. Thus the passenger path is delayed, and the edge $(v_{e_1}, t)$ is in the cut and accounts for $\delta \cdot w(P)$ delay. If $j = l$, the passenger path is maintained; no other edge related to $P$ traverses the cut, either. If $1 \leq j < l$, $\{v_{e_{j+1}}, \ldots, v_{e_l}\} \subset \bar{S}$, the passenger path is dropped. In this case, the edge $(v_{e_j}, v_{e_{j+1}})$ traverses the cut, contributing to the cut’s cost with $(T - \delta) \cdot w(P)$. In total, a weight $T \cdot w(P)$ is in the cut for passenger path $P$, matching the weighted arrival delay for this case. Finally, we are left with the case of a source delayed passenger path $P = \{e_1, \ldots, e_l\}$ dropped at its initial leg. In this case, $e_1 \in \bar{S}$, and the edge $(s, e_1)$ traverses the cut, correctly accounting for the arrival delay $T w(P)$ of dropping $P$. \qed
Restricting delay policies: maintaining source punctual passenger paths

In every day’s life, dropping source punctual passenger paths seems unfair. Thus, we address the special delay policy which explicitly prohibits doing so, but still allows to drop source delayed passenger paths and delay source punctual passenger paths by $\delta$.

This setting can be solved by a reduction to a minimum cut problem as well. To that aim, we again interpret the partition of the nodes by the cut as a partition of the legs into a set of legs waiting for delayed passenger paths and a set of legs which depart as scheduled.

The main idea is to introduce infinite-weight edges between the different leg-nodes boarded by a source punctual passenger path. In that way, such an edge will never traverse a minimum cut, and thus guarantee that a source punctual passenger path is maintained when delayed. Furthermore, we make use of the construction for source punctual passenger paths of Theorem 3.7 also for $k > 3$. Again, the edge connecting the passenger path-node $v_P$ to $t \in S$ accounts for the cost of delaying the source punctual passenger path $P$.

More precisely, given an instance $Q$ of the binary delay management problem with the restricted delay policy that source punctual passenger paths cannot be dropped, we build an instance $I(Q)$ of the minimum cost directed $s$-$t$-cut problem on a weighted graph $H = (U, F)$ with $s, t \in U$, and weight function $F \rightarrow \mathbb{N}$ for the edges as follows.

Let $U = \{v_e : e \in E\} \cup \{s, t\} \cup \{v_P : p \in \mathcal{P}\}$. For each source punctual passenger path $P \in \mathcal{P}, P = \{e_0, e_1, \ldots, e_l\}, e_i \in E, D(P) = 0$, we introduce the edges $(v_{e_i}, v_{e_{i+1}}), i \in \{0, \ldots, l-1\}$ each of weight $c(v_{e_i}, v_{e_{i+1}}) = \infty$ into $F$. Let $v_P \in U$ be the node corresponding to $P$. We introduce the edges $(v_{e_i}, v_P), i \in \{0, \ldots, l\}$ each with weight $c(v_{e_i}, v_P) = \infty$, and an edge $(v_P, t)$ with weight $c(v_P, t) = \delta \cdot w(P)$. For each source delayed passenger path $P = \{e_0, \ldots, e_l\}, P \in \mathcal{P}, D(P) = \delta$, let $v_P$ be the node corresponding to the considered passenger path. We introduce an edge $(s, t)$ of weight $c(s, t) = \delta \cdot w(P)$, the edges $(v_P, v_{e_i}), i \in \{0, \ldots, l\}$, each edge with weight $c(v_P, v_{e_i}) = \infty$, and, finally, an edge $(s, v_P)$ with weight $c(s, v_P) = (T - \delta) \cdot w(P)$.

**Lemma 3.11.** Consider an instance $Q$ of the binary delay management problem, and a restrict the set of feasible delay policies to those
which do not drop source punctual passenger paths. Restricted to these policies, the minimum total passenger delay for $Q$ is equal to the cost of the minimum weighted directed $s$-$t$-cut $[S, \bar{S}]$ of the instance $I(Q)$. One optimal delay policy is to make all legs corresponding to nodes in $S$ wait and all legs corresponding to nodes in $\bar{S}$ depart on time.

Proof. Since we can drop all source delayed passenger paths by setting $S = \{s\}$, and that $s$ has no outgoing edges of infinite weight, at least one non-infinite cut exists. Next, we show that source punctual passenger paths can never be dropped. Finite weight edges appear only from and to nodes $v_P$, and between $s$ and $t$. The infinite weight edges ensure the desired consistency. Assume that the source punctual passenger path $P = \{e_0, \ldots, e_l\} \in P, D(P) = 0$, has an arrival delay. Then, there exists an $e_i \in P : v_{e_i} \in S$. Because of the infinite weight edges $(v_{e_i}, v_{e_{i+1}}), \ldots (v_{e_{l-1}}, v_{e_l})$, no such edge can traverse the minimum directed cut. Hence, the passenger path will not be dropped. Note that source punctual passenger paths can be delayed, as the infinite weight edges traverse the cut backwards from $\bar{S}$ to $S$ and are hence not counted in the objective.

We show that the cost of each non-infinite weight cut is equal to the delay occurring if all nodes corresponding to legs in $S$ wait, and all nodes corresponding to legs in $\bar{S}$ depart on time. Each source punctual passenger path $P = \{e_0, \ldots, e_l\}$, has $v_P \in S$ if at least one $v_{e_i} \in S$, i.e., if the passenger path is delayed. If this were not the case, at least one infinite weight edge would traverse the cut. Since $v_P \in S$, the edge $(v_P, t)$ traverses the cut, increasing its cost by the weighted arrival delay of $P$. If the passenger path is not delayed, then $v_P \in \bar{S}$, and accordingly no edge related to $P$ traverses the cut in any direction. For each source delayed passenger path $P = \{e_0, \ldots, e_l\}, e_i \in E, D(P) = \delta$, per construction $(s, t)$ traverses the cut, adding the weighted arrival delay when maintaining $P$ to the cut’s cost. Further, assume one of the legs boarded by $P$ departs on time, and hence the (source delayed) passenger path is dropped. Then, the node $v_P$ is in $\bar{S}$, as an infinite weight edge traverses the cut otherwise. Correspondingly, the edge $(s, v_P)$ traverses the cut, increasing the cut’s costs for this passenger path to $T \cdot w(P)$, which corresponds to the weighted arrival delay of $P$ if $P$ is dropped. If the passenger path is not dropped, then $v_P \in S$, and no other edge traverses the cut.

Chapter 3. Offline delay management

Giving an importance to each connection

The previous results also show a viable approach for the binary delay management problem where the importance of a connection is not given as a result of the passenger paths using it, but with an abstract cost for each connection. The abstract cost of the connection, fixed and independent from the number of passengers actually using it, occurs if the connection is not maintained. High cost connections thus represent connections of high importance. In a similar way, legs waiting for delayed feeder legs, thus guaranteeing the connection, incur in a fixed, leg dependent cost. The goal is to minimize the sum of the overall costs given by the delay policy, that is, of the dropped connections combined with the waiting costs of the legs. This setting has a natural interpretation: some train services are more important than other (as, for instance, intercity trains with respect to regional trains), and the cost of dropping a connection or making a train wait reflects this importance. Making an important train wait costs more than making a less important train wait (where by cost we mean an abstract measure, not necessarily money). Breaking connections from less important trains to more important trains might also have a lower cost than the opposite. Thus, less important trains incur in a smaller cost if they wait for transfers from a more important trains. On the other hand, given that many less important trains are late, the cost of dropping all connections might outweigh the cost of making the important train wait.

This setting can be represented with our model for the binary delay management problem: for each connection between two legs, we introduce a passenger path of length two using the considered legs, and set the dropping cost of a passenger path to the importance of the connection. For each leg, we introduce a passenger path of length one using it, with waiting costs equal to the leg waiting costs. Finally, the delayed legs induce a source delay on the above passenger paths. Now, it is easy to see that the cost structure specified above can be used for to the construction of the minimum-cut graph. Thus, this problem can be solved to optimality in polynomial time by solving a minimum cost directed $s$-$t$-cut problem. Note that this setting cannot be extended in a straight-forward manner to multiple delay sizes, since with this construction the cut can only distinguish between legs that run as scheduled and legs that wait. As we assume zero transfer time, all transfers between legs that wait are guaranteed.
3.2. Minimum-cut based approaches

3.2.3 Fixed parameter tractability

The approaches based on the minimum directed cut allow for a fixed-parameter tractable algorithm when the number transfers per passenger paths is not bounded by two.

**Theorem 3.12.** Given an instance of the binary delay management problem with \( p \) source punctual passenger paths, its minimum cost delay policy can be found in \( O(2^p \cdot (|E| + |P|)^2 \log(|E| + |P|)) \) time. Thus, the binary delay management problem is fixed parameter tractable in the number of passenger paths.

**Proof.** The general idea of the fixed parameter tractable algorithm is the following. First, we note that as soon as we have decided which source punctual passenger paths we drop, the problem turns easy. Indeed, this implies that all other source punctual passenger paths are maintained. By removing the passenger paths which we decided to drop from the instance, this setting can be solved as shown in Theorem 3.11. Thus, the following approach is a fixed parameter algorithm. Enumerate all \( 2^p \) subsets of source punctual passenger paths. For each subset \( S \), solve the given instance restricted on the passenger paths \( P \setminus S \) with the delay policy of maintaining all source punctual passenger paths with the minimum cut based approach of Section 3.2.2. Let \( \pi \) be the resulting delay policy, and \( W \) its total passenger delay. Extend \( \pi \) by dropping all source punctual passenger paths \( S \), set \( W := W + T \sum_{p \in S} w(p) \). Note that the delay policy may be such that some of the passenger paths in \( S \) are nevertheless maintained, and thus \( W \) overestimates the costs of such a delay policy. If \( W \) is lower than the cost \( W^* \) of the best policy \( \pi^* \) seen so far, assign \( W^* := W, \pi^* := \pi \). Since we enumerate all possible configurations of dropped source punctual passenger paths, and for each such configuration we find the optimal solution, we also enumerate the global optimum solution. This approach has a running time of \( O(2^p \cdot (|E| + |P|)^2 \log(|E| + |P|)) \), since the computation of each of the \( 2^p \) minimum cuts requires \( O((|E| + |P|)^2 \log(|E| + |P|)) \) time. \( \square \)
3.3 Dynamic programming for a railway corridor

This section describes a solution approach for the binary delay management problem with a network topology of a railway corridor. For this restricted setting, we present an enumeration tree for all feasible solutions, and show that equivalent subtrees of that enumeration tree can be pruned. This observation leads to a polynomial time algorithm for the finding the delay policy giving the minimum total passenger delay. Finally, we show that the pruned enumeration tree algorithm boils down to a dynamic programming algorithm.

3.3.1 Enumerating all delay policies

The problem instances of the binary delay management problem considered in this section have two specific characteristics. First, we restrict the railway network to a corridor. Since a corridor corresponds to a directed path in the railway network, this implies that $G$ is a simple path. Thus, we denote the ordered set of nodes by $V = (v_1, \ldots, v_{m+1})$, with $v_1 < \ldots < v_{m+1}$, and the set of edges by $E = (e_1, \ldots, e_m)$, with $e_i = (v_i, v_{i+1})$.

Second, we restrict the size of the source delay to $\delta = 1$. Thus each passenger path $P \in \mathcal{P}$ has a binary source delay $D(P) \in \{0, 1\}$. Our analysis below only uses the fact that all non-zero source delays are identical; we use 0-1 delays only because of ease of exposition. As before, these delays imply that a leg serving as a connection either departs as scheduled, or it waits for all delayed passenger paths.

Therefore, the decision whether a leg waits for delayed passenger paths or not is modeled by the following wait-depart decision variable:

$$x_i = \begin{cases} 
1 & \text{if leg } e_i \text{ waits one time unit at station } v_i, \\
0 & \text{if leg } e_i \text{ departs from station } v_i \text{ as scheduled.}
\end{cases}$$

As before, we consider the total passenger delay objective, and the set of feasible delay policies can be represented as the set of vectors $(x_1, \ldots, x_m) \in \{0, 1\}^m$.

In the following, we describe a binary tree $H$ of height $m$ which enumerates the possible configurations for the decision variables $x_i$. 

3.3. Dynamic programming for a railway corridor

In $H$, branching between the levels $i - 1$ and $i$ represents choosing a value for the variable $x_i$. A node at level $i$ in $H$ has a label $\langle x_1, \ldots, x_i \rangle$, where $x_1, \ldots, x_i$ are the decisions taken on the unique path from the root node to that node. The root node itself has the empty label $\langle \rangle$. So, the label $\langle x_1, \ldots, x_i \rangle$ immediately contains the partial solution at the node, and a leaf node $\langle x_1, \ldots, x_m \rangle$ at level $m$ represents a solution to the model.

At node $\langle x_1, \ldots, x_i \rangle$, wait-depart decisions have been taken for the legs $e_1, \ldots, e_i$. We keep track of the impact of these decisions through the following functions of the nodes which we discuss in the following (we omit the node label brackets here to improve readability):

\[
A(x_1, \ldots, x_i) = \left\{ P \in \mathcal{P} \mid s(P) \leq v_i < t(P), P \text{ not dropped by } x_1, \ldots, x_i \right\}
\]

\[
D(x_1, \ldots, x_i) = \sum_{P \in \mathcal{P} \text{ dropped by } x_1, \ldots, x_i} T \cdot w(P) + \sum_{P \in \mathcal{P} : t(P) \leq v_i, P \text{ delayed by } x_1, \ldots, x_i} w(P),
\]

\[
D_m(x_1, \ldots, x_m) = D(x_1, \ldots, x_m) + \sum_{P \in A(x_1, \ldots, x_m)} w(P) \cdot x_m.
\]

The set of active passenger paths $A(x_1, \ldots, x_i)$ contains all passenger paths $P$ that can board leg $e_i$, given the decisions at node $\langle x_1, \ldots, x_i \rangle$. In a sense complementary, $D(x_1, \ldots, x_i)$ contains the already accumulated weighted delay caused so far by the decisions at node $\langle x_1, \ldots, x_i \rangle$. So, at any level $i$ in $H$, each passenger path $P \in \mathcal{P}$ with $s(P) \leq v_i$ is either contained in $A(x_1, \ldots, x_i)$, or its weighted arrival delay is accounted for in $D(x_1, \ldots, x_i)$. Note that $A(x_1, \ldots, x_i)$ contains passenger paths $P$ with $t(P) = v_{i+1}$, although the arrival delay for such passenger paths is known when the decisions $x_1, \ldots, x_i$ have been taken. In particular, this fact means that the active passenger path set $A(x_1, \ldots, x_m)$ at a leaf node may be non-empty. Therefore, $D_m(x_1, \ldots, x_m)$ accounts for the weighted arrival delay at $v_{m+1}$ of all passenger paths $P$ on leg $e_m$, given the decisions at node $\langle x_1, \ldots, x_m \rangle$. An optimal solution to our model is then represented by a leaf node $\langle x_1, \ldots, x_m \rangle^*$ that attains a minimum value $D_m(x_1, \ldots, x_m)$.

For the root node $\langle \rangle$, we set $D(\langle \rangle) = 0, A(\langle \rangle) = \emptyset$. Below, we specify the initialization for the child nodes $\langle 0 \rangle$ and $\langle 1 \rangle$ of the root
node $\langle \rangle$. Next, we describe a general child node $\langle x_1, \ldots, x_i, x_{i+1} \rangle$ with parent node $\langle x_1, \ldots, x_i \rangle$. Since the values of $A(x_1, \ldots, x_{i+1})$ and $D(x_1, \ldots, x_{i+1})$ depend on the parent’s values and on the values of the decisions $x_{i+1}$ and $x_i$, we distinguish between the four possible combinations for a child node $\langle \ldots, x_i, x_{i+1} \rangle$.

**Initialization of $\langle 0 \rangle$ and $\langle 1 \rangle$**

$$A(0) = \{ P \in \mathcal{P} | s(P) = v_1, D(P) = 0 \}, \quad D(0) = \sum_{P \in \mathcal{P} : s(P) = v_1, D(P) = 1} T \cdot w(P),$$

$$A(1) = \{ P \in \mathcal{P} | s(P) = v_1 \}, \quad D(1) = 0.$$ 

$\langle \ldots, 0, 0 \rangle$ nodes

$$A(x_1, \ldots, x_i, x_{i+1}) = A(x_1, \ldots, x_i) \setminus \{ P \in \mathcal{P} | t(P) = v_{i+1} \}$$

$$\cup \{ P \in \mathcal{P} | s(P) = v_{i+1}, D(P) = 0 \},$$

$$D(x_1, \ldots, x_i, x_{i+1}) = D(x_1, \ldots, x_i) + \sum_{P \in \mathcal{P} : s(P) = v_{i+1}, D(P) = 1} T \cdot w(P).$$

$\langle \ldots, 0, 1 \rangle$ nodes

$$A(x_1, \ldots, x_i, x_{i+1}) = A(x_1, \ldots, x_i) \setminus \{ P \in \mathcal{P} | t(P) = v_{i+1} \}$$

$$\cup \{ P \in \mathcal{P} | s(P) = v_{i+1} \},$$

$$D(x_1, \ldots, x_i, x_{i+1}) = D(x_1, \ldots, x_i).$$

$\langle \ldots, 1, 0 \rangle$ nodes

$$A(x_1, \ldots, x_i, x_{i+1}) = \{ P \in \mathcal{P} | s(P) = v_{i+1}, D(P) = 0 \},$$

$$D(x_1, \ldots, x_i, x_{i+1}) = D(x_1, \ldots, x_i) + \sum_{P \in A(x_1, \ldots, x_i) : t(P) > v_{i+1}} T \cdot w(P)$$

$$+ \sum_{P \in \mathcal{P} : s(P) = v_{i+1}, D(P) = 1} T \cdot w(P) + \sum_{P \in A(x_1, \ldots, x_i) : t(P) = v_{i+1}} w(P).$$
3.3. Dynamic programming for a railway corridor

$\langle \ldots, 1, 1 \rangle$ nodes

\[
A(x_1, \ldots, x_i, x_{i+1}) = A(x_1, \ldots, x_i) \setminus \{ P \in \mathcal{P} | t(P) = v_{i+1} \}
\cup \{ P \in \mathcal{P} | s(P) = v_{i+1} \},
\]

\[
D(x_1, \ldots, x_i, x_{i+1}) = D(x_1, \ldots, x_i) + \sum_{P \in A(x_1, \ldots, x_i): t(P) = v_{i+1}} w(P).
\]

As an example, and since it is a special case, we briefly discuss the case of a $\langle \ldots, 1, 0 \rangle$ node. Because $x_i = 1$ and $x_{i+1} = 0$, all passenger paths in $A(x_1, \ldots, x_i)$ which do not have destination $v_{i+1}$ cannot transfer to leg $e_{i+1}$ and are dropped, facing a weighted arrival delay of $T \cdot w(P)$. Leg $e_{i+1}$ is also missed by all passenger paths $P \in \mathcal{P}$ with origin in $v_{i+1}$ and source delay $D(P) = 1$, so these passenger paths also face a weighted arrival delay of $T \cdot w(P)$. Further, since $x_i = 1$, all passenger paths $P \in A(x_1, \ldots, x_i)$ having node $v_{i+1}$ as destination have a weighted arrival delay of $w(P)$. Finally, the only passenger paths $P$ that can board leg $e_{i+1}$ are those with origin in $v_{i+1}$ and having $D(P) = 0$.

3.3.2 Pruning equivalent subtrees

An enumeration tree node $\langle \ldots, 1, 0 \rangle$ implies that all passenger paths wanting to transfer from leg $e_i$ to leg $e_{i+1}$ will miss their connection. Therefore, none of the passenger paths in $A(x_1, \ldots, x_i)$ will make it into $A(x_1, \ldots, x_i, x_{i+1})$. Thus, the subtree rooted at node $\langle x_1, \ldots, x_{i+1} \rangle$ is in a sense independent of the decisions $x_1, \ldots, x_i$ taken before. The following Lemmas show that this independence allows us to prune the enumeration tree significantly.

**Lemma 3.13.** For an enumeration subtree rooted at node with label $\langle x_1, \ldots, x_i \rangle$, let

\[
D_{\langle x_1, \ldots, x_i, x_{i+1}, \ldots, x_{i+k} \rangle} := D(x_1, \ldots, x_{i+k}) - D(x_1, \ldots, x_i),
\]

that is, the accumulated weighted arrival delay in the node of the subtree $\langle x_{i+1}, \ldots, x_{i+k} \rangle$. Any two enumeration subtrees rooted at nodes labeled $\langle x_1, \ldots, x_{i-1}, x_i \rangle$ and $\langle x_1', \ldots, x_{i-1}', x_i' \rangle$ with $x_{i-1} = x_{i-1}' = 1$ and $x_i = x_i' = 0$ are equivalent in the sense that for $k = 1, \ldots, m - i$,

\[
D_{\langle x_1, \ldots, x_i \rangle}(x_{i+1}, \ldots, x_{i+k}) = D_{\langle x_1', \ldots, x_i' \rangle}(x_{i+1}, \ldots, x_{i+k}).
\]
Proof. Note that the subtree root \( \langle x_1, \ldots, x_i \rangle \) plays the role of the empty labeled root \( \langle \rangle \) in the original tree. We use the shorthand notation \( \langle \ldots, 1, 0 \rangle \) for some node with a label \( \langle x_1, \ldots, x_{i-2}, 1, 0 \rangle \). From the construction of \( \langle \ldots, 1, 0 \rangle \) nodes, it is clear that

\[
A(\ldots, 1, 0) = \{ P \in \mathcal{P} | s(P) = v_i, D(P) = 0 \}.
\]

So, \( A(\ldots, 1, 0) \) is the same for each node \( \langle \ldots, 1, 0 \rangle \). Therefore, both the enumeration subtrees rooted at \( \langle x_1, \ldots, x_i \rangle \) and \( \langle x'_1, \ldots, x'_i \rangle \) have the same passenger path set \( A(\ldots, 1, 0) \), and will thus accumulate the same weighted arrival delay up to subtree node \( \langle x_{i+1}, \ldots, x_{i+k} \rangle \). \( \square \)

Lemma 3.14. Of all subtrees rooted at \( \langle x_1, \ldots, x_{i+1} \rangle \), with \( x_i = 1, x_{i+1} = 0 \), it is sufficient to explore the single subtree with root

\[
\langle x_1, \ldots, x_{i+1} \rangle^* = \operatorname{argmin}_{\langle x_1, \ldots, x_{i+1} \rangle} \left\{ D(x_1, \ldots, x_{i+1}) \mid x_i = 1, x_{i+1} = 0 \right\}
\]

(3.3.1)

Proof. The result follows from Lemma 3.13 with \( k = m - i \), which implies that the minimum between \( D(x_1, \ldots, x_{i+1}, x_{i+2}, \ldots, x_m) \) and \( D(x'_1, \ldots, x'_{i+1}, x_{i+2}, \ldots, x_m) \) is determined by the minimum value of \( D(x_1, \ldots, x_{i+1}) \) and \( D(x'_1, \ldots, x'_{i+1}) \). \( \square \)

Because of this pruning, the number of nodes in the enumeration tree can be reduced significantly from \( O(2^m) \) to \( O(m^3) \), as is stated by the following Lemma. Moreover, this leads to an overall worst case running time of \( O(m^5) \) for the pruned enumeration tree algorithm.

Lemma 3.15. The pruned enumeration tree has \( O(m^3) \) nodes.

Proof. In order to count the number of nodes in the enumeration tree, we define the following variables:

\[
N^0(i) \quad \text{The number of nodes } \langle x_1, \ldots, x_{i-1}, 0 \rangle \text{ at level } i.
\]

\[
N^1(i) \quad \text{The number of nodes } \langle x_1, \ldots, x_{i-1}, 1 \rangle \text{ at level } i.
\]

From the definition of the initialization phase, it follows that \( N^0(1) = N^1(1) = 1 \). At level \( i + 1 \), a child node \( \langle \ldots, 1 \rangle \) is created for every parent node \( \langle \ldots, 0 \rangle \), and also for every parent node \( \langle \ldots, 1 \rangle \). Therefore, \( N^1(i + 1) = N^0(i) + N^1(i) \). Further, for every parent node
3.3. Dynamic programming for a railway corridor

⟨..., 0⟩, a child node ⟨..., 0⟩ is created. But, by Lemma 3.14, one single child node ⟨..., 0⟩ is created for all parent nodes ⟨..., 1⟩. This yields $N^0(i + 1) = N^0(i) + 1$. Solving the recurrences

$$\begin{align*}
N^0(1) &= 1, & N^1(1) &= 1, \\
N^0(i + 1) &= N^0(i) + 1, & N^1(i + 1) &= N^0(i) + N^1(i),
\end{align*}$$

gives $N^0(i) = i$ and $N^1(i) = \frac{1}{2}i^2 - \frac{1}{2}i + 1$. So, the total number of nodes at level $i$ is $O(m^2)$. With $m$ levels, the pruned enumeration tree has $O(m^3)$ nodes in total. □

Theorem 3.16. The binary delay management problem on a corridor with passenger paths having binary source delay $\{0, 1\}$ can be solved in $O(m^5)$ time.

Proof. The pruned enumeration tree has $O(m^3)$ nodes, and at each node at most $|P| = O(m^2)$ passenger paths have to be evaluated to compute the functions $A(\cdot)$ and $D(\cdot)$. □

3.3.3 A dynamic programming view

This section shows that the pruned enumeration tree algorithm can also be written as a dynamic program. The dynamic program is stated with the following two partial solution value functions:

For all $k \leq i$ :

$$z_{10}(i, k) = \min_{(x_1, \ldots, x_i)} \left\{ D(x_1, \ldots, x_i) \mid x_{k-1} = 1, x_k, \ldots, x_i = 0 \right\}.$$  

For all $k < j \leq i$ :

$$z_{101}(i, j, k) = \min_{(x_1, \ldots, x_i)} \left\{ D(x_1, \ldots, x_i) \mid \begin{array}{l}
x_{k-1} = 1, \\
x_k, \ldots, x_{j-1} = 0, \\
x_j, \ldots, x_i = 1.
\end{array} \right\}$$

The function $z_{10}(i, k)$ denotes the minimum value of all partial solutions where anything can have happened until leg $e_{k-2}$, leg $e_{k-1}$ waits for delayed passenger paths, and all subsequent legs $e_k, \ldots, e_i$ depart as scheduled. Similarly, the function $z_{101}(i, j, k)$ denotes the
minimum value of all partial solutions where anything can have happened until leg $e_k-2$, leg $e_k-1$ waits for delayed passenger paths, the consecutive legs $e_k, \ldots, e_{j-1}$ depart as scheduled, and all consecutive legs $e_j, \ldots, e_i$ again wait for delayed passenger paths. The subscripts for the functions $z_{10}$ and $z_{101}$ stand for the structure of the last significant events in the function’s partial solutions.

Similar to $A(x_1, \ldots, x_i)$, we define the set of passenger paths $P \in \mathcal{P}$ that are active in the best partial solution represented by $z_{101}(i, j, k)$ as follows:

$$A(i, j, k) = \left\{ P \in \mathcal{P} \mid s(P) \geq v_k, t(P) > v_i, \ P \text{ not dropped in } z_{101}(i, j, k) \right\}.$$ 

Using $A(i, j, k)$, the recursion formulas for the dynamic program for the two functions $z_{10}(i, k)$ and $z_{101}(i, j, k)$ are expressed as follows:

**If** $i + 1 > k$:

$$z_{10}(i + 1, k) = z_{10}(i, k) + \sum_{P \in \mathcal{P}: \ s(P)=v_{i+1}, \ D(P)=1} T \cdot w(P) \text{ if } i + 1 > k.$$

**If** $i + 1 = k$:

$$z_{10}(i + 1, k) = \min_{j, k'} \left\{ z_{101}(i, j, k') + \sum_{P \in A(i, j, k'): \ t(P)>v_{i+1}} T \cdot w(P) \right. \left. + \sum_{P \in \mathcal{P}: \ s(P)=v_{i+1}, \ D(P)=1} T \cdot w(P) + \sum_{P \in A(i, j, k'): \ t(P)=v_{i+1}} w(P) \right\}.$$

**If** $i + 1 > j$:

$$z_{101}(i + 1, j, k) = z_{101}(i, j, k) + \sum_{P \in A(i, j, k): \ t(P)=v_{i+1}} w(P).$$
If $i + 1 = j$:

$$z_{101}(i + 1, j, k) = z_{10}(i, k)$$

These recursions uniquely correspond to the four cases for the updates of the functions $D(x_1, \ldots, x_{i+1})$ for the enumeration tree node $\langle x_1, \ldots, x_{i+1} \rangle$. In particular, the case $i + 1 = k$ for $z_{10}(i + 1, k)$ corresponds to the pruning of the enumeration tree when only a single $\langle \ldots, 1, 0 \rangle$ node is created at level $i + 1$.

### 3.3.4 Extensions of the algorithm

From the analysis above, it is clear that the algorithm also works for source delays in $\{0, \delta\}$. A closer inspection of the dynamic program’s recursion formulas discloses that it can also be carried out backwards. Finally, the dynamic programming algorithm can be extended for the case of $K$ source delay categories, i.e., $D(P) \in \{\delta_1, \ldots, \delta_K\}$. In this case, it is necessary to keep track of any increase of waiting time of the legs along the corridor; every decrease in waiting time between two subsequent legs acts as a drop between the two legs. Thus, this extension comes at the cost of a worst case running time of $m^{O(K)}$.

### 3.4 Train services with intermediate stops

The results presented so far have a wider influence than only the delay management problem without intermediate stops. Indeed, in the following we adapt the previous results to show hardness for the binary delay management problem with intermediate stops and polynomial time solvability of restricted variants thereof.

#### 3.4.1 Proof of hardness

First, we define the decision version of delay management with intermediate stops.
Chapter 3. Offline delay management

Definition: Decision binary delay management problem with intermediate stops.

Instance: An instance to the binary delay management problem with intermediate stops, \( d \in \mathbb{N} \).

Question: Is there a delay policy for the given instance such that the total passenger delay is less than or equal to \( d \)?

Corollary 3.17 (of Theorem 3.2). The decision binary delay management problem with intermediate stops on a series-parallel network with unit weight passenger paths having at most two transfers and train services having at most one intermediate stop is \( NP \)-complete.

Proof. We prove Corollary 3.17 by adapting the reduction of Theorem 3.2. Note that our aim is to reduce the number of transfers from three to two. For each extended edge \((u, u_e, v_e, v)\), we build one multiple-leg train services from the construction with single-leg train services of Figure 3.4.1. First, let \( r_u = \{(B_u, C_e), (C_e, D_e)\} \) be served by a single train, thus having an intermediate stop. Since these two legs build a directed path from \( B_u \) to \( D_e \), it is a legal train service. Moreover, the character of the gadget is unaltered: the passenger path \( P_{u_e} \) is the only passenger path transferring in \( C_e \), and it is never dropped in \( C_e \). Indeed, it either reaches \( C_e \) on time or has already been dropped in \( B_u \) if the leg \((A_u, B_u)\) of the train service waits for the source delayed passenger path \( P_u \). Second, let \( r_v = \{(B_v, D_e), (D_e, E_e)\} \). Again, \( r_v \) is a legal train service with one intermediate stop. As the train serving \( r_v \) never waits in \( B_v \), the train service arrives at \( D_e \) on time, thus maintaining the character of the gadget. All remaining edges are served each by a different train. Now, the maximum number of transfers any passenger path performs is two, since the only two types of passenger paths transferring three times in the reduction of Theorem 3.2 now use one train service for traveling two of their four legs. Furthermore, as the contraction to a series-parallel graph does not modify any edges, this construction can be contracted to a series-parallel graph as well. 

Note that the extension applies also to Theorem 3.4, as the edges \( e_1 \) and \( e_2 \) can be served by a single train. This keeps the number of transfers of all source delayed passenger paths starting in \( C \) in the original construction at one, and introduces at most one intermediate stop per train service.
3.4. Train services with intermediate stops

P
1
P
2
P
α
P
3
P
v
1
B
u
A
u
B
v
A
v
P
1
C
E
D
P
2
P
v
2
P
v
3
P
v
e

Figure 3.4.1: Start configuration for Theorem 3.17. Red dashed lines are source delayed, green solid lines source punctual passenger paths. Thick passenger paths represent passenger paths of weight $M$.

3.4.2 Polynomial-time solvable cases

The minimum cut approach can be extended to the binary delay management problem with intermediate stops, given that source punctual passenger paths transfer at most once.

The following construction allows to find an optimal solution to the delay management problem for instances where each source punctual passenger path transfers between train services at most once. Similarly to the previous minimum-cut reductions, given an instance $Q$ of the binary delay management problem with intermediate stops, where source punctual passenger paths transfer at most once, we construct an instance $I(Q)$ of the minimum cost directed $s$-$t$-cut problem on a graph $H = (U,F)$, with $s, t \in U$ and the cost function $c : F \mapsto \mathbb{N}$.

We map each leg $e \in E$ to a node $v_e \in U$, add two nodes $s$ and $t$ to $U$. For each train service $R = \{e_1, e_2, \ldots, e_r\}$ we introduce edges $(v_{e_i}, v_{e_{i+1}}), 1 \leq i < r$ with weight $c(v_{e_i}, v_{e_{i+1}}) = \infty$. As soon as $e_i \in S$, these edges prevent the nodes $e_k, i < k \leq r$ from being in $\bar{S}$, providing the required consistency that train services that wait cannot catch up on their delay. Hence, each train service in $\mathcal{R}$ can wait for delayed passenger paths, but once it did so, it will never be on time again.

For the passenger paths we introduce the following constructions. The described constructions are for general passenger paths. Specific passenger paths, as passenger paths using only one leg, need not the
complete construction. For the sake of uniformity, we describe just the general setting. For each source punctual passenger paths \( P = \{e_1, \ldots, e_l\} \), transferring between train services at most once, we introduce a node \( v_P \), an edge \((v_P, t)\) with weight \( c(v_P, t) = \delta w(P) \), and edges \((v_{e_i}, v_P)\), \( \forall e_i \in P \) with weight \( c(v_{e_i}, v_P) = +\infty \). This construction enforces \( v_P \) to be in \( S \) as soon as at least one of the nodes \( v_{e_i}, e_i \in P \) is in \( S \). The weight of the edge \((v_P, t)\) accounts for the cost of delaying \( P \) by \( \delta \) time units. A source punctual passenger path can only miss a connection if it transfers between two different train services. For this to happen, the first train service arrives at the station where the transfer occurs with a delay, and the second train service departs from that station as scheduled. Let \( e_j \) be the last leg of \( P \) traveled with the first train service, and \( e_k \) be the first leg of \( P \) traveled with the second train service. To account for the additional delay of dropping \( P \), we introduce the edge \((v_{e_j}, v_{e_k})\) with weight \( c(v_{e_j}, v_{e_k}) = (T - \delta)w(P) \).

For each source delayed passenger path \( P = \{e_1, \ldots, e_l\} \) we introduce an edge \((s, t)\) with weight \( c(s, t) = \delta w(P) \) accounting for the inevitable source delay of such passenger paths. Furthermore, we introduce a node \( v_P \), an edge \((s, v_P)\) with weight \( c(s, v_P) = (T - \delta)w(P) \), and the edges \((v_P, v_{e_i})\), \( \forall e_i \in P \) with weight \( c(v_P, v_{e_i}) = +\infty \). This construction aims at enforcing \( v_P \) into \( \bar{S} \) as soon as at least one of the nodes \( v_{e_i} \) is in \( \bar{S} \), thus forcing the edge \((s, v_P)\) with weight \((T - \delta)w(P)\) through the cut.

**Theorem 3.18.** Consider an instance \( Q \) of the binary delay management problem with intermediate stops with source punctual passenger paths transferring at most once, and the instance \( I(Q) \) of the minimum directed \( s-t \)-cut problem on the weighted directed graph \( H = (U, F) \) derived from \( Q \). The cost of the minimum \( s-t \)-cut \([S, \bar{S}]\) on \( H \) is equal to the total passenger delay of a minimum cost delay policy for \( Q \).

**Proof.** A non-infinite weight cut exists, as the partition \( S = \{s\}, \bar{S} = U \setminus \{s\} \) has weight \( T \cdot \sum_{P \in \mathcal{P}: \delta(P) = \delta} w(P) \): For each source delayed passenger path \( P \), its edge \((s, t)\) contributes (weighted) \( \delta \) to the weight of the cut, and each passenger path’s edge \((s, v_P)\) contributes an additional (weighted) \((T - \delta)\) to it. Next, we show that the consistency is ensured for each train service. Since a cut of non-infinite weight exists, no infinite weight edge traverses the cut in any
optimal solution. Thus, since all the nodes representing the legs of a train service are connected in $H$ through a directed path with infinite weight edges, as soon as one of these nodes is in $S$, all subsequent nodes representing legs of that train service are also in $S$, reflecting that train services cannot catch up on their delay. For each source delayed passenger path $P$, an edge $(s, t)$ accounts for the source delay of the passenger path. If and only if one of the passenger path’s legs travels as scheduled, and thus the corresponding node is in $\bar{S}$, by construction also the node $v_P$ is in $\bar{S}$, as an infinite weight edge would traverse the cut otherwise. Therefore, the edge $(s, v_P)$ traverses the cut, bringing the contribution of the dropped passenger path $P$ to the cut’s weight to $T_w(P)$. For each source punctual passenger path $P$, as soon as one of the boarded legs waits, the corresponding node $v_P$ is in $S$, as an infinite weight edge would traverse the cut otherwise. Thus, the edge $(v_P, t)$ traverses the cut, contributing $\delta w(P)$ to the cut’s weight. A source punctual passenger path may only be dropped if it uses two train services, the last boarded leg $e_j$ of the first train service waits and the first boarded leg $e_k$ of the second train service departs as scheduled. If this happens, the edge $(v_{e_j}, v_{e_k})$ traverses the cut, contributing an additional $(T - \delta)w(P)$ to the cut’s weight. Thus, the overall weight for this case is $T_w(P)$, correctly representing the cost of dropping $P$.

3.5 Binary delay management problem with slack times

In the previous sections, we showed which aspects make the binary delay management problem hard to solve and how to handle some cases in polynomial time. In this section, we address the added difficulty of slack times (see Section 2.3.3) for the binary delay management problem. In this section, we address two problem variants: first, we show hardness for the binary delay management problem with slack times without intermediate stops, where each leg has either slack time zero or a slack time $\delta$ equal, in value, to the source delays. Second, we analyze restricted networks and passenger path topologies for instances of the binary delay management problem with slack times and intermediate stops, where legs have different slack times, and give simple polynomial-time algorithms for solving
these restricted problem variants to optimality.

### 3.5.1 Proof of hardness

First, we define the decision variant of the binary delay management problem with slack times and no intermediate stops as follows:

**Definition:** Decision binary delay management problem with binary slack times.

**Instance:** An instance of the binary delay management problem with slack times and without intermediate stops, with \( S(e) \in \{0, \delta\} \forall e \in E, d \in \mathbb{N} \).

**Question:** Does a delay policy for the instance exist, such that the total passenger delay does not exceed \( d \)?

In Section 3.1.1, we showed that the decision binary delay management problem is \( \mathcal{NP} \)-complete on series-parallel networks with passenger paths having at most three transfers. Here, we show that by including binary slack times, delay management is \( \mathcal{NP} \)-complete already with passenger paths transferring at most twice. Without slack times, this variant can be solved in polynomial time (Section 3.2). Furthermore, we show that the hardness reduction can be adapted to a railway network with the topology of a railway corridor. This variant is polynomial-time solvable without slack times as well (see Section 3.3). These two facts give a strong indication that slack times do indeed make delay management harder to solve.

The hardness results for the decision binary delay management problem with binary slack times rely on the hardness of Maximum Directed Acyclic Cut problem. Although it is well known that Maximum Directed Cut is \( \mathcal{NP} \)-complete, the hardness proof in [47] does not create an acyclic graph.

**Definition:** Maximum Directed Acyclic Cut

**Instance:** An unweighted directed acyclic graph \( G = (V, E) \), \( K \in \mathbb{N} \).

**Question:** Does a partition of \( V \) into two disjoint sets \( V_1, V_2, V = V_1 \cup V_2 \) exist, such that the number of edges traversing the partition from \( V_1 \) to \( V_2 \) is greater than or equal to \( K \)?

**Lemma 3.19.** Maximum Directed Acyclic Cut is \( \mathcal{NP} \)-complete.

**Proof.** The problem is in \( \mathcal{NP} \), as any solution can be verified in polynomial time. We prove \( \mathcal{NP} \)-hardness by reduction from Maxi-
3.5. Binary delay management problem with slack times

The maximum Unweighted Directed Cut [20, Problem ND16]: given a directed graph \( G = (V, E) \), \( |V| = n \), \( |E| = m \) and a positive integer \( K \in \mathbb{N} \), is there a partition of \( V \) into two disjoint sets \( V_1, V_2 \), \( V = V_1 \cup V_2 \), such that the number of edges traversing the cut from \( V_1 \) to \( V_2 \) is at least \( K \)?

We first build a maximum directed acyclic cut instance \( G' = (V', E') \) using edge weights \( c' \) as follows. For each node \( v_i \in V \), we build a structure of five nodes, \( \{v^1_i, v^2_i, v^3_i, v^4_i, v^5_i\} \), connected by four edges \( (v^j_i, v^{j+1}_i), j \in \{1, \ldots, 4\} \), with weight \( c'(v^j_i, v^{j+1}_i) = m \). We show later how to make this construction unweighted. Clearly, at most two non-consecutive edges of each structure can traverse the cut. By setting their weights to \( m \) we enforce that two of these edges actually do traverse the cut. For each edge \( e = (v_i, v_j) \in E \), we insert the unweighted edge \( (v^2_i, v^4_j) \) into \( E' \). The reduction is polynomial in space and time: we have \( 5n \) nodes and \( 4n + m \) edges, and the graph can be constructed efficiently.

The graph \( G \) has a maximum cut \( V_1, V_2 \) of size \( K \) if and only if \( G' \) has a maximum cut \( V'_1, V'_2 \) of size \( 2nm + K \).

Assume \( G \) has a cut \( [V_1, V_2] \) of size \( k \). For \( v_i \in V \), we set \( \{v^2_i, v^4_i\} \subset V'_1 \) and \( \{v^1_i, v^3_i, v^5_i\} \subset V'_2 \) if \( v_i \in V_1 \), and the opposite if \( v_i \in V_2 \). In each of the \( n \) constructed structures of \( G' \) exactly two edges of weight \( m \) traverse the cut. Further, for each edge \( (v_i, v_j) \) traversing the cut in \( G \) there is the edge \( (v^2_i, v^4_j) \) traversing it in \( G' \). Hence, the cut in \( G' \) has size \( 2nm + k \).

Assume \( G' \) has a cut of size \( 2nm + k \). We build a cut of size \( k \) in \( G \) by setting \( v_i \in V_1 \) if \( v^2_i \in V'_1 \) and \( v_i \in V_2 \) if \( v^2_i \in V'_2 \). Again, each construction in \( G' \) contributes weight \( 2m \) to the cut’s size. Each of the additional \( k \) edges traversing the cut in \( G' \) now induces an edge traversing the cut in \( G \). Consider an arbitrary edge \( (v^2_i, v^4_j) \) which is in the cut in \( G' \), so \( v^4_j \in V'_2 \). Per construction, \( v_i \in V_1 \). If \( (v_i, v_j) \) is not in the cut in \( G \), we must have \( v^2_j \in V'_1 \). With this configuration, though, only one of the edges with weight \( m \) for \( v_j \) can traverse the cut \([V'_1, V'_2]\), namely either \((v^2_j, v^3_j)\) or \((v^3_j, v^4_j)\). As in an optimal solution for \( G' \) two edges of weight \( m \) per structure must traverse the cut, the choice \( v^2_j \in V'_1 \) is not optimal, hence \( (v_i, v_j) \) is in \( G' \)’s cut.

The above reduction also works for unweighted graphs \( G' \). In that case we introduce, for each structure, \( m \) parallel paths of length two between even-numbered nodes instead of the edges of weight \( m \),
Chapter 3. Offline delay management

The decision binary delay management problem with binary slack times is \( \mathcal{NP} \)-complete with unit weight passenger paths having each at most two transfers.

Proof. The proof is by reduction from Maximum Directed Acyclic Cut. Since the delay of each passenger path can be efficiently computed for any given delay policy, the problem is in \( \mathcal{NP} \). Given a maximum directed acyclic cut instance \( G^C = (V^C, E^C) \), we build a decision binary delay management problem with binary slack times as follows, see Figure 3.5.1. For every \( v \in V^C \), we introduce an leg \( f_v \in E \) without slack time. For each edge \( e = (u, v) \in E^C \), we introduce a leg \( g_e \) from \( f_u \) to \( f_v \) having slack time equal to \( \delta \). Further, for each edge \( e = (u, v) \in E^C \), we introduce four passenger paths: The source delayed passenger path \( P_e^u = (f_u, g_e, f_v) \) of unit weight, and the set \( P_e \) of three parallel source punctual passenger paths \( \{f_u\} \) of unit weight. Note that each outgoing edge \( (u, v) \in E^C \) induces the three passenger paths \( P_e^u \) on \( f_u \).

We set \( \delta = 1 \) and \( T = 4 \), and ask for a delay policy inducing a total passenger delay of at most \( d = mT - K \delta = 4m - K \). There is a direct correspondence of a delay policy in \( G \) to a cut \([V_1, V_2] \) \( \Box \)
in $G^C$: if $f_u$ waits, $u \in V_1$, otherwise $u \in V_2$. It remains to be shown that there is a cut of size at least $K$ if and only if there is a delay policy with total delay at most $d$. To this end, it is sufficient to analyze the delay caused by the passenger paths $P_e$ and $P^{u}_e$ for the different policies. If $f_u$ does not wait, $P_e$ is dropped and all of $P^{u}_e$ have zero arrival delay. So, independent of $f_v$, these passenger paths together contribute $T$ to the objective. If both $f_u$ and $f_v$ wait, the passenger paths contribute $4\delta = T$ to the objective, as all passenger paths have an arrival delay $\delta$. If $f_u$ waits and $f_v$ departs as scheduled, the passenger paths contribute $3\delta$ to the objective. Now, $G^C$ has a maximum directed cut of size $K$ if and only if the binary delay management problem with binary slack times has a delay policy causing $4m - K = d$ total passenger delay. This delay is achieved allowing $K$ passenger paths of type $P_e$ to have zero arrival delay. Using the described correspondence between a cut in $G^C$ and a delay policy in $G$, for every edge $e$ of $G^C$ there is a contribution of 3 units to the total delay if $e$ crosses the cut, and of 4 units otherwise. □

In contrast to Lemma 3.11, no source punctual passenger paths are dropped in the above construction. Note that the reduction can be adapted to any $T = k\delta$ by introducing $k - 1$ parallel passenger paths $P^{u}_e$ per edge $e$ instead of three. The special case $k = 1$ is also feasible, but it is unclear how it should be interpreted. Furthermore, the possibility to dynamically reroute passenger paths does not influence the construction of Theorem 3.20, since the first and the last edge of the passenger paths $P_e$ cannot be changed. This observation allows us to simplify the network topology even further:

**Corollary 3.21.** The decision binary delay management problem, with binary slack times is $\text{NP}$-complete with unit weight passenger paths, even if the network forms a corridor.

**Proof.** Since $G$ is acyclic, we can order the nodes topologically, i.e., for every edge $(u_i, u_j) \in E$ we have $i < j$. Now replace the edges $g_e$ by edges $g_i$ connecting $f_{u_i}$ with $f_{u_{i+1}}$, with slack time $\delta$. Additionally, each passenger path $P_e$ for $e = (u, v)$ now uses all legs on the path between (and including) $f_u$ and $f_v$. Since all $g_i$ have slack time equal to $\delta$, a passenger path of type $P_e$ can only be dropped at its first leg $f_u$. □

In general, this proof yields passenger paths with an arbitrary
Chapter 3. Offline delay management

number of transfers. The structure of the created instance has the following natural interpretation. The legs without slack times stand for real train services, the legs with slack times stand for transfers at the stations. The first and last activity of a passenger path are on a real train service. Given that the slack time for transferring at stations is large enough, there is no propagation of the delays.

The first of the previous reductions showed hardness for a small number of transfers per passenger path, the second one for a simple network topology. In the following, we give a reduction which combines these two aspects by using a slightly more complex network structure.

Corollary 3.22. The decision binary delay management problem, with binary slack times is \( NP \)-complete on a series-parallel network with unit weight passenger paths each having at most two transfers.

Proof. The reduction is from Maximum Directed Cut on a graph \( G^C = (V^C, E^C) \), \(|V^C| = n, |E^C| = m\). The main idea is to use the construction of Lemma 3.19 to make \( G^C \) acyclic, and then translate the resulting Maximum Directed Acyclic Cut instance to a delay management instance by means of the construction given in Theorem 3.20.

Without loss of generality, we can place each node of type \( v^1_i \) from Lemma 3.19 in the node partition \( V_1 \), and all nodes of type \( v^5_i \) in \( V_2 \). This fact allows us to simplify some of the passenger paths in the problem instance of the decision binary delay management with binary slack times, schematically depicted in Figure 3.5.2 for an edge \((u, v) \in V^C\). The source delayed passenger path \( P^4_v \) should continue to one more edge without slack: as the passenger path uses the edge alone, this is superfluous. Similarly the passenger path \( P^1_v \) should start at an earlier edge with slack time \( \delta \), but is never dropped there since the passenger path uses that leg alone.

As in the proof of Theorem 3.2, the network \( G' \) can be made series-parallel by contracting nodes with the same functionality, see Figure 3.5.3. Thus, the network consists of only 7 nodes, and bundles of parallel edges. Note that each bundle of edges with slack time \( \delta \) can be contracted to a single edge with slack time \( \delta \), since every leg with slack time can wait, and still arrive at its destination punctually. Each passenger path now uses its original edges without slack times, and the contracted edges with slack time. Furthermore, each passenger
3.5. Binary delay management problem with slack times

Figure 3.5.2: The delay management instance with slack times resulting from an edge \((u, v)\) in the Maximum Cut instance \(G^C\). Directed edges represent legs. Dotted legs have slack time \(\delta\), plain ones have no slack time. Undirected lines represent passenger paths. Solid green passenger paths are source punctual, dashed red ones are source delayed.

Figure 3.5.3: The contraction of the corresponding nodes (here shown for the network above) leads to three bundles of parallel edges for edges with no slack time.

path still interacts only with the relevant passenger paths on the same dedicated edges without slack time as before the contraction.

The proof of Corollary 3.22 might suggest that re-routing passenger paths simplifies the problem. This is not the case, since the route of each passenger path can be made unique by the techniques sketched in Section 3.1.3. In this case, we only identify the second and the fifth node. Note that the edge with slack time between these two nodes can be removed as well, since the passenger paths \(P_{u,v}\) (see Figure 3.5.3) can be rerouted through other existing edges between the two nodes. Except for this latter type of passenger path, all other passenger paths have a unique route, and each route for \(P_{u,v}\) induces the same costs.
3.5.2 Polynomially solvable cases

Although the general setting on the line is $\mathcal{NP}$-hard, some variants of the binary delay management problem with slack times can be solved efficiently by simple strategies. Below we describe three such variants.

All passengers to the terminal station of a single train service

In contrast to the models analyzed so far, we consider the binary delay management problem with slack times and with intermediate stops. However, we restrict the set of train services to cardinality one. Thus, one train service serves the complete railway network, and the railway network $G$ builds a simple path, a railway corridor. Again, we order the set of nodes in $G$ as $V = \{v_1, \cdots, v_{m+1}\}$, with $v_1 < \cdots < v_{m+1}$ and denote the set of edges by $E = \{e_1, \cdots, e_m\}$ with $e_i = (v_i, v_{i+1})$. Each leg $e \in E$ of this train service may have a nonnegative slack time $S(e) \in \mathbb{Q}_0^+$, each of different size. Thus, the train service travels from node $v_1$ to the node $v_{m+1}$, the train service’s source and terminal, respectively, and performs intermediate stops at each of the nodes $v_i$, $i \in \{2, \cdots, m\}$. Because of this special network structure, once a passenger path has boarded a leg, it cannot connect to legs served by other trains, as there are none. Thus, a passenger path can either be dropped at its origin node, or it reaches its destination node, possibly with some delay.

We assume that all passenger paths $P \in \mathcal{P}$ have the terminal node of the train service as destination. Again, we can interpret this setting as passengers traveling to the city center on an urban rail line. Finally, we drop the restriction on the source delays to be binary, and allow arbitrary rational numbers for the source delays of the passenger paths. Thus, the source delay of each passenger path is specified by $\mathcal{D} : P \rightarrow \mathbb{Q}_0^+$. We refer to this model as the delay management problem with slack times and all passengers to a unique destination on a single train service.

**Theorem 3.23.** The delay management problem with slack times and all passengers to a unique destination on a single train service can be solved in polynomial time.
Proof. This problem can be solved by the following pedal-to-the-metal strategy. The driver a priori fixes a target delay at the terminal stop $v_{m+1}$, exhausts all slack times, and drives at maximum velocity to achieve that target delay.

Given a target arrival delay $\delta'$, the delay policy $\pi$ corresponding to the pedal-to-the-metal strategy is computed in a backward fashion, starting from the train service’s terminal node. Let $\delta_i'$ be the delay of the train service when arriving at node $v_i$. The delay of the train service at node $v_{i-1}$ is computed as $\delta_{i-1}' = \delta_i' + S((v_{i-1}, v_i))$. Any optimal policy must use all the available slack, since using less slack only results in dropping more passenger paths. Thus, the policy $\pi$ is optimal for the arrival delay $\delta'$.

To see which values are relevant as target arrival delays $\delta'$ at the terminal node, we start considering the arrival delay $\delta' = 0$ (recall that legs cannot travel earlier than scheduled). Let $l(\delta')$ be the minimum time by which a passenger path missed the train service aiming at a target arrival delay $\delta'$. If the train service had waited $l(\delta')$ longer, thus targeting for an arrival delay $\delta' + l(\delta')$, only this passenger path would additionally be maintained. Hence, we analyze the target arrival delay $\delta' + l(\delta')$ next. Since the values $\delta' \in (\delta_i', \delta_i' + l(\delta_i'))$ result in dropping the same passenger paths as for $\delta_i$ but increase the arrival delay, they need not be considered. This procedure can then be iterated. As at most $|\{(P \in \mathcal{P} : D(P) > 0)\}|$ passenger paths can lead to a different arrival delay, only polynomially many solutions need to be evaluated, and we can pick the best one.

All passengers to the terminal with single trains on a rooted tree

The method above to create a delay policy can be extended to the following more complex case. Here, we consider the railway network to be a rooted in-tree, that is, a rooted tree where all edges point towards the root. Thus, there is a directed (graph-theoretic) path between each node of the tree and the root node. Each leg of the railway network is served by one train service, and each train service serves exactly one of these legs. Similar to the previous setting, we allow general rational slack times for each leg. Finally, we restrict the passenger paths to travel to the root node of the tree; thus, each passenger path has the root node of the tree as destination node. Furthermore, each passenger path has a (possibly zero) nonnegative, rational source de-
lay. With this configuration, the passenger paths must connect to a different train service at each intermediate node between their origin node and their destination node, the root node.

We point out that it makes no sense to drop a passenger path at a node different from its origin node. By doing that, we would drop all passenger paths on board of that leg which start before that connection. But then, we may as well have the preceding legs wait less, drop the considered path at its source station, and maintain the connections for the passenger paths which can board the leg with these delays. Since we then drop less connections than before, or in extreme cases the same connections, this delay policy is optimal on a corridor. It is also optimal for a tree, as the argument holds for every inbound leg. Hence, the pedal-to-the-metal strategy for the delay management problem with slack times and all passengers to a unique destination on a single train service can be applied to this problem as well.

When $G$ is a general tree, this kind of strategy also works if all passenger paths start at a common stop, or if all passengers travel in the same direction through a common leg (but not with a common station only).

**Passenger paths with at most one transfer and binary slack times**

A final restricted setting which can be solved in polynomial time is as follows. We consider the binary delay management problem with slack times and without intermediate stops. In a similar fashion as for the binary delay management problem, we restrict the number of transfers per passenger path to at most one. Furthermore, we again restrict the slack times to be binary. Hence, each leg $e \in E$ has slack time $S(e) \in \{0, \delta\}$, where $\delta$ is the size of source delays.

This setting can be reduced to a minimum-cut problem. The reduction basically enumerates all relevant settings for passenger paths and slack times (and there are a limited number of them), and follows the ideas of the previous minimum-cut approaches. Thus, we again map the legs of the delay management instance to nodes in the minimum cost directed $s$-$t$-cut problem. We specify the weights which we introduce for each passenger path in the following.

In general, it cannot be harmful for a leg that waits to use up all the available slack time. Indeed, this strategy causes the minimal possi-
ble arrival delay for that leg, thus causing the smallest possible arrival
delay for the passenger paths ending with that leg, and potentially
 guaranteeing some more connections. In the setting considered here,
a leg with slack time can always wait for its late passenger paths and
arrive at destination as scheduled. Indeed, by doing so no delayed
passenger path is dropped, and no passenger path arrives later than
scheduled at the end of this leg because of that decision. Therefore,
it is sufficient to consider schedules where all legs with slack time
wait. Moreover, this readily implies that any passenger path with no
transfers using a leg with slack time has zero arrival delay, regardless
of it source delay. For passenger paths which use no legs with slack
times, we have already shown a reduction for paths with at most two
transfers (Lemma 3.7), which naturally also applies for shorter paths.
Thus, we can use this construction for passenger paths which do not
wish to board legs with slack times, and extend the idea of the reduc-
tion, to passenger paths using legs with slack times.

We remain with considering passenger paths with one transfer
which wish to board one or two legs with slack time. Similar to the
case with no transfers, any passenger path using two legs with slack
times cause zero delay, as both legs can wait for late passenger paths
and arrive at destination punctually. Now, consider $P = \{e_1, e_2\}$
with $S(e_1) = \delta, S(e_2) = 0$. Here, only the arrival delay of $e_2$ is
relevant, and applies to $P$ no matter its source delay, as the passenger
path $P$ arrives at $e_2$’s source station as scheduled ($e_1$ waits). Hence,
the passenger path’s contribution to the objective is determined by
whether $e_2$ waits or not for other passenger paths. A source punctual
passenger path $P = \{e_1, e_2\}$ with $S(e_1) = 0, S(e_2) = \delta$ always has
an arrival delay of zero time units, as the source punctual passenger
path cannot miss $e_1$ and the slack time of $e_2$ guarantees its punctual
arrival at its destination. However, a source delayed passenger path $P$
with the same leg configuration misses its connection if $e_1$ does not
wait. In all other cases, $P$ reaches its destination punctually.

Using the construction of Lemma 3.7 for the passenger paths us-
ing solely legs without slack time, we handle the remaining cases as
follows: for each $e \in E, S(e) = \delta$, we introduce an edge $(s, v_e)$
with weight $c(s, v_e) = \infty$. Such edges guarantee a delay policy
which makes all legs with slack times wait. For all passenger paths
$P = \{e_1, e_2\} \in \mathcal{P}$ with $S(e_1) = \delta, S(e_2) = 0$, we introduce an edge
$(v_{e_2}, t)$ with weight $c(v_{e_2}, t) = \delta w(P)$, which accounts for the pas-
senger path’s delay if $e_2$ waits. Finally, for all source delayed passenger paths $P = \{e_1, e_2\} \in \mathcal{P}$, $D(P) = 0$ with $S(e_1) = 0, S(e_2) = \delta$, we introduce an edge $(s, v_{e_1})$ with weight $c(s, v_{e_1}) = Tw(P)$, which accounts for the passenger path’s delay if $e_1$ does not wait. For all other configurations, we introduce no edges. The given construction leads to the following theorem:

**Theorem 3.24.** The minimum delay policy for the binary delay management problem with slack times and without intermediate stops, with binary slack times, can be solved in polynomial time by reduction to a minimum cost directed s-t-cut problem, given that passenger paths using legs with slack time transfer at most once, and that passenger paths using legs with no slack time transfer at most twice.
Chapter 4

Renting skis and managing delays online

[As they watch the plane take off without them:]
Sucre: What do we do now?
Michael: We run.
Prison Break, Season 1, Episode 22.

In this chapter, we consider the online version of delay management. As previously stated, delay management is inherently an online problem. Indeed, delays occur over time and delaying the decision process in order to wait for all occurring delays is not an option. Therefore, an online setting is well suited for delay management (with the only exception, perhaps, that contrary to the canonical online setting, exponential-time decision rules are not practicable in real life).

Given the lack of research an online delay management, we generally consider the basic case of a single corridor, where either all consecutive legs are served by a single train, or where each leg is served by a different train. This choice is further motivated by the fact that delay management is computationally easier on corridors (see Chapter 3.3). Moreover, each large railway network typically decomposes into several important corridors. However, we also consider tree-like network topologies with a restrictive assumption on the passenger paths.

A delay policy specifies which legs wait for delayed passenger paths. Our goal is to find a delay policy that minimizes the total pas-
senger delay, without knowing the complete set of source delayed passenger paths beforehand, but receiving them sequentially. This imperfect knowledge reflects the online character of the problem.

Chapter outline and summary of results  Many of the results of this chapter are joint work with other persons. The results on the generalized ski rental problem are joint work with Riko Jacob, Marc Nunkesser, Leon Peeters, Michael Schachtebeck, Anita Schöbel and Peter Widmayer. The remaining results are joint work or were inspired by fruitful discussions with with Riko Jacob, Leon Peeters and Peter Widmayer.

The next section (4.1) describes the different variants of the ski rental problems we consider, and gives a short introduction to the competitive analysis. Next, we define the delay management model used in the online setting (Section 4.2.1). We show the strong connection between different settings of online delay management on a railway corridor and the well known classical ski rental problem \[33\] and extensions thereof in Section 4.2.2.

In Section 4.3, we propose and analyze a family of 2-competitive online algorithms for various settings of delay management on a railway corridor, and show the tightness of some competitive analyses. Moreover, we extend the family of online algorithm to more complex tree-like railway networks with the restriction that all passengers travel to a terminal node.

Section 4.4 presents lower bounds on the competitive ratio for any online algorithm on a railway corridor. We prove that no online algorithm for the single train serving the corridor can have a competitive ratio of less than \(\Phi \approx 1.618\) (the golden ratio). We also show that no algorithm can have a competitive ratio less than 2 when considering multiple trains serving a long corridor.

Section 4.5 describes some extensions, and discusses the limitations of our ski rental inspired approaches. The relation to the ski rental problem allows to apply the analysis of the randomized algorithm against an oblivious adversary for the ski rental problem \[32\], thus providing a best-possible competitive ratio for a randomized algorithm of \(\frac{e}{e-1}\). Next, we address the objective of minimizing the additional passenger delay. For this objective, we show that no deterministic online algorithm can have bounded competitive ratio. Remarkably, the only strategy that does not have an infinite competitive
ratio is the trivial strategy of waiting for any delayed passenger, and departing on time otherwise. Further, we show the limitations of our ski rental approaches for dealing with less restricted problem settings.

4.1 Ski rental and online algorithms

In the classical ski rental problem, a skier wants to go to ski. Since the skier initially owns no skis, she faces two options: she can either buy skis at a fixed price $b$, or rent skis each day (also at a fixed price of 1 per day). Obviously, buying skis is more expensive than renting skis, but if the skier goes skiing for a long time, buying is worthwhile (given that she can dispose of the skis for free). Unfortunately, she does not know how many times she’ll ski, only that at some point she might realize that she no longer enjoys it and will never ski again. However, maybe she will enjoy skiing so much, that she’ll never stop. If the skier knew how long she was skiing, the decision would be trivial: she would buy the skis on the first day if renting them for the whole period were more expensive than buying the skis. The online character of the problem renders such a decision impossible: the skier is faced with the choice of renting the skis until she eventually buys them or does not enjoy skiing anymore. Each day she is faced with the option, she does not know if she will still be skiing the day after.

In general, an online algorithm is presented a request sequence. Thus, an instance of an online problem is characterized by a possibly infinite request sequence $I = (r_1, r_2, \ldots, r_n)$. An online algorithm $A$ serves the request sequence with an answering sequence $A = (a_1, a_2, \ldots, a_n)$, with the following properties. Each answer $a_i$ is taken based only on the knowledge of the requests $r_j, j \leq i$ and its previous answers $a_j, j < i$. Hence, the online algorithm has no knowledge about future requests, or even if further requests exist, and must serve each request before the next one is presented. Furthermore, the answer $a_j$ must be consistent with the previously given answers, in the sense that the online algorithm may not revoke decisions $a_i, i < j$ when serving request $r_j$. The cost of an online solution is a function of the request sequence $I$ and the answer sequence $A$. Let $C_A(I)$ be the cost of the solution of the online algorithm $A$ on the input sequence $I$, and let $C_{OPT}(I)$ be the cost of an optimal offline solution for the sequence $I$. By optimal offline we mean the solution of a clairvoyant algorithm which knows the complete sequence $I$ in
advance and can thus serve it optimally.

For ski rental, the request sequence is built by the days where the skier goes to ski. Here, such a request sequence is somewhat strange, since the problem is completely specified by the daily renting cost and the buying cost of the skis. Indeed, an online algorithm must decide how high the overall renting costs will be before the skier buys the skis, and the decision is only dependent on the renting costs and on the buying cost of the skis. Formally, the ski rental problem is defined as follows: given a daily renting cost of 1 and a buying cost \( b \in \mathbb{Q}_0^+ \) for the skis, find the overall paid renting costs \( R = 1 \cdot k, k \in \mathbb{N} \) before the skier buys the skis, where \( k \) is the number of days the skier should rent the skis before buying them. An instance is specified by \( I = (b, k^*) \), where \( k^* \in \mathbb{N} \) is the number of days the skier will go skiing and is disclosed online (the request sequence). The costs are defined as follows. As the online algorithm does not know \( k^* \) in advance, the cost of an online algorithm renting the skis for the first \( k \) days are:

\[
C_A(I) = \begin{cases} 
  k^* & \text{if } k^* \leq k \\
  k + b & \text{otherwise}
\end{cases}
\]

The optimal offline algorithm, on the other hand, knows the number of days \( k^* \) the skier will be skiing. Thus, it can compute an optimal solution by comparing the cost of renting the skis for the \( k^* \)-day period, or buying them on the first day. Thus, the costs of the optimal offline algorithm are defined as:

\[
C_{OPT}(I) = \min\{k^*, b\}
\]

The quality of an online algorithm is measured by means of competitive analysis. An online algorithm for a minimization problem is called \( c \)-competitive if for all possible input sequences \( I \), \( C_A(I) \leq c \cdot C_{OPT}(I) + c' \), where \( c' \) is a constant independent from the instance. The algorithm is called strictly \( c \)-competitive if \( c' = 0 \). The value \( c \) is called the competitive ratio of the online algorithm. Online problems are often seen as a game between two players: an online algorithm and an adversary. The adversary determines the input sequence in such a way that, given the decisions sequentially taken by the online algorithm, it maximizes the competitive ratio.
A well known deterministic online algorithm for the ski rental problem is to start by renting the skis for a few days, and to buy them as soon as the overall renting costs match or exceed the cost of buying the skis \[33\]. Hence, the online algorithm selects \( R = \lceil b - 1 \rceil \) and thus rents for the first \( k = \lceil b - 1 \rceil - 1 \) days. This deterministic strategy is known to be strictly 2-competitive for the case of continuous renting costs (that is, when the renting cost each day are infinitesimally small), and is best-possible. The proof for the discrete case defined above is given as an example in the following, and leads to a slightly better competitive ratio.

Let \( A_x \) be the online algorithm which rents the skis as long as the overall renting costs, including those of the considered day, are strictly less than \( x \). Thus, the day the overall renting costs would reach or exceed \( x \), the online algorithm buys the skis. Because of the discrete nature of the problem, the values of \( x \) relevant for the analysis can be limited to integral numbers. Indeed, for \( s \in \mathbb{N}^+ \) and \( y \in \mathbb{R} \), all algorithms \( A_y \) with \( y \in (s, (s + 1)] \) behave as \( A_{(s+1)} \). All of these algorithms first rent for \( s \) days and buy on day \((s + 1)\).

The algorithm \[33\] above is thus specified as \( A_b \), with the restriction that \( b \in \mathbb{N}^+ \). Now, we consider the ratio \( \frac{C_{A_b}(I)}{C_{\text{OPT}}(I)} \) for all possible input sequences; the maximum ratio achieved over all possible sequences \( I \) is the competitive ratio. First, we consider the case \( b > k^* \) where it is worthwhile renting for the whole \( k^* \)-day period. Now, since \( b > k^* \), the number of days \( k \) the online algorithm rents the skis is greater or equal to \( k^* \). Hence, the ratio is \( \frac{C_{A_b}(I)}{C_{\text{OPT}}(I)} = \frac{k^*}{k^*} = 1 \), the online algorithms performs the right choice. Consider the case \( b \leq k^* \) where it is worthwhile buying the skis at the beginning of the \( k^* \)-day period. Because of the case analysis, \( k < k^* \) and the online algorithm buys the skis on day \((k + 1)\) after having paid \( b - 1 \) for renting the skis. Thus, for this case the ratio evaluates to \( \frac{C_{A_b}(I)}{C_{\text{OPT}}(I)} = \frac{k+b+1}{b} - \frac{b-1+b}{b} = 2 - \frac{1}{b} \), which leads to a competitive ratio of \( 2 - \frac{1}{b} \). Thus, the evil adversary maximizes the competitive ratio by choosing \( k^* = b \).

The choice \( x = b \) is best possible. Assume \( x = b + \sigma, \sigma \in \mathbb{N}^+ \). Then, the adversary can choose \( k^* = b+\sigma \). The optimal solution is to buy the skis. The online algorithm rents the skis for \( k^* - 1 \) days and buys them on day \( k^* \). This strategy leads to a ratio of \( \frac{C_{A_{b+\sigma}}(I)}{C_{\text{OPT}}(I)} = \frac{2b+\sigma-1}{b} \geq 2 \). Thus, the choice of \( x \) larger than \( b \) is suboptimal.
Assume \( x = b - \sigma, \sigma \in \mathbb{N}^+ \). Then, the adversary can choose \( k^* = b - \sigma \), which makes it worthwhile to rent. However, the online algorithm first rents for \( k^* - 1 \) days, then buys the skis, incurring in a total cost of \( b - \sigma - 1 + b = 2b - \sigma - 1 = 2(b - \sigma) + \sigma - 1 \), compared to the costs \( k^* = b - \sigma \) of renting the skis for the whole period. Hence, the ratio of the two solutions is \( \frac{2(b-\sigma)+\sigma-1}{b-\sigma} = 2 + \frac{\sigma-1}{b-\sigma} \geq 2 \), and the choice of \( x \) smaller than \( b \) is again suboptimal.

### 4.1.1 Discounted ski rental problem

In order to show the relation between online delay management and ski rental problems, we introduce the following slight variant of the latter.

Since the skier always rents the skis at the same shop, the nice shop owner gives her a discount on buying the skis. The discount is proportional to the amount of money the skier has already paid for renting skis, and the proportionality factor is \( \alpha \), and is known to the skier. Moreover, as the request for rented skis varies from day to day, the ski’s renting price is variable. Each day, the skier is told that day’s renting cost for the skis. Thus, on the \( i \)-th day, the renting price of the skis is \( r_i \), whereas the price for buying the skis is fixed at \( b \) each day.

We call this problem the *discounted ski rental problem*. An input sequence is specified by \( I = (b, \alpha, \{r_i : i \in \mathbb{N}\}) \), where \( b \in \mathbb{Q}_0^+ \) is the cost for buying the skis, \( \alpha \in \mathbb{Q}_0^+ \) the discount factor, and the renting costs \( r_i \in \mathbb{Q}^+ \) are disclosed in an online fashion.

For the moment neglecting the competitive ratio of the following approach, it is clear that the online algorithm for the classical ski rental problem can also be applied to the discounted ski rental problem. In that case, the skier buys the skis on the day when the overall renting costs would exceed the actual discounted price of buying the skis. We give the competitive analysis of the discounted ski rental problem indirectly in Section 4.3.1 by giving a one-to-one correspondence to the delay management problem (Section 4.2.2) and showing the competitive ratio of a ski rental like approach (corresponding to the algorithm sketched above) for a general setting of delay management.
4.1. The generalized \( k \)-day ski rental problem

In the following, we analyze a further generalization of the ski rental problem in the online setting, which we call the generalized \( k \)-day ski rental problem. The problem variant shares some traits with the discounted ski rental problem.

The generalized \( k \)-day ski rental problem is defined as follows. As before, a skier wants to go skiing, but she initially does not own any skis, nor does she know for how long she’ll enjoy skiing. However, the skier knows she’ll be skiing for at most \( k \) days (imagine she booked a \( k \)-day holiday), so she’ll be faced with the rent-or-buy decision at most \( k \) times. She can either buy the skis at a cost of 1, fixed for the whole \( k \) day period (for simplicity, we have scaled the buying cost to one), or rent the skis. As in the discounted ski rental problem, the renting costs are variable and can thus differ from day to day. In this variant, however, rather than being told a renting price each day, the whole list of renting prices is disclosed on the first day in the following way. On the first day, the skier is given a list of monotone increasing prices \( \{x_1, \ldots, x_k\} \) which specify the overall cost of renting the skis up to each of the \( k \) days.

An input sequence for the generalized \( k \)-day ski rental problem is thus specified by \( I = (\{x_1, \ldots, x_k\}, i^*) \), where \( x_i \in \mathbb{Q}_0^+ \) are the overall renting costs up to day \( i \), the cost of the skis is always fixed to 1 and \( i^* \in \mathbb{N} \) is the actual number of days the skier will ski, and is the information disclosed online.

As for the classical ski rental problem, the offline strategy is simple: if the skier knows for how long she’ll be skiing, she can compute the overall cost of renting the skis. If the renting cost exceeds the cost of buying the skis, she buys the skis on the first day, otherwise she rents them for the whole period. As usual in a competitive analysis the adversary is aware of this and will choose the renting costs such that it is cheaper to buy the skis than to rent them for the whole \( k \) day period (otherwise the problem is trivial). If the skier rents the skis up to day \( i \) and then buys them, she incurs a cost of \( x_i + 1 \). Note that the classical ski rental online algorithm of Section 4.1 is 2-competitive also for this case.

There is another interpretation for the variable renting costs: we can assume that instead of having a time horizon of \( k \) days, we have a horizon of \( x_k \) days (here, we need to perform some scaling on the
renting costs and on the price of the skis; we neglect this step as it is simply a scaling factor). This horizon is subdivided into \( k \) intervals, the first \([x_0 = 0, x_1]\), the last \([x_{k-1}, x_k]\). At the beginning of the \( i \)-th interval, the skier knows she’ll be skiing until the end of that interval, but doesn’t know if she’ll continue skiing afterwards. Hence, she can either rent the skis for that interval at a cost of \( x_i - x_{i-1} \), thus overall paying \( x_i \) to rent the skis up to day \( x_i \), or buy them at a price of 1. Thus, if the skier rents up to the \( i \)-th interval and then buys the skis, she incurs a cost of \( x_i + 1 \).

We provide a competitive analysis of the generalized \( k \)-day ski rental problem, which we present as a game between the skier and the adversary. First, note that w.l.o.g. the adversary chooses \( x_i \leq 1 \) for \( 1 \leq i \leq k - 1 \), because with \( x_i > 1 \) the skier must buy in step \( i \) in order to guarantee a competitive ratio of 2.

For the choice of \( x_k \), we do a case analysis: If the skier ever considers the value \( x_k \), she has already paid \( x_{k-1} \) for renting the skis and now chooses the minimum of \( x_k \) and \( 1 + x_{k-1} \). In this case the competitive ratio \( \hat{C}_k \) equals

\[
\hat{C}_k = \sup_{x_k} \frac{\min\{1 + x_{k-1}, x_k\}}{\min\{1, x_k\}}.
\]

For \( x_k \leq 1 \) we get \( \hat{C}_k = 1 \). For \( 1 \leq x_k \leq 1 + x_{k-1} \), we get \( \hat{C}_k \leq 1 + x_{k-1} \). Finally, for \( x_k \geq 1 + x_{k-1} \) we get \( \hat{C}_k = 1 + x_{k-1} \). Therefore, as the adversary wants to maximize \( \hat{C}_k \), w.l.o.g. we let \( x_k = 1 + x_{k-1} \).

As in all ski rental problems, the optimal strategy of the adversary is simply to let the skier ski up to the day where she buys the skis. After this day, the adversary stops the skier from skiing. This strategy is optimal, because the only alternative would be to stop the skier before she has bought the skis, which, from the above observation, must be at a day \( k' < k \). In this case, as \( x_{k'} < 1 \), the adversary cannot achieve a competitive ratio larger than one.

As the adversarial strategy is clear, all cases can be parameterized by the day \( i \in \{1, \ldots, k\} \) when the skier buys the skis. With the convention \( x_0 = 0 \), and letting \( C_i \) be the competitive ratio which results from buying the skis on day \( i \), we have

\[
C_i = \begin{cases} 
\frac{1 + x_{i-1}}{x_i} & \text{for } 1 \leq i < k \\
1 + x_{k-1} & \text{for } i = k
\end{cases}
\]
4.1. Ski rental and online algorithms

Therefore, if the skier plays optimally against the adversary, she can achieve a competitive ratio $C$ by choosing the day $i$ when she buys the skis such as to minimize, for the given renting costs, the value of $C_i$. On the other hand, the adversary chooses the renting costs in such a way that the minimum competitive ratio $C_j, j \in \{1, \cdots, k\}$ is maximum. A closer inspection of the ratios yields that the maximum is attained if all ratios $C_j, j \in \{1, \cdots, k\}$ are equal. Thus,

$$C = \frac{1}{x_1} = \frac{1 + x_1}{x_2} = \cdots = 1 + x_{k-1} \quad (4.1.1)$$

Repeatedly substituting for $x_i$ in (4.1.1) yields $x_i = x_1 + x_1^2 + \cdots + x_1^i$ for $i = \{1, \ldots, k-1\}$. By substituting this expression in the equation $1 = x_1(1 + x_{k-1})$, we obtain the following equation containing the geometric series:

$$1 = x_1 + x_1^2 + \cdots + x_1^k \quad (4.1.2)$$

As already stated, $0 \leq x_i < 1, i \in \{1, \cdots, k-1\}$. Thus, we are interested only in the real roots of equation (4.1.1) in the interval $[0, 1]$. It is easy to see that such a root exists and is unique.

For $k = 2$ the equation is $x_2^2 + x_1 - 1 = 0$. The positive root of this equation is $\frac{-1}{2} + \frac{1}{2} \sqrt{5} = \frac{1}{\Phi} = 0.618 \cdots$, which is known as the golden ratio conjugate. Thus, the achievable competitive ratio is $\frac{1}{x_1} = \Phi$, the golden ratio.

For $k \to \infty$, equation (4.1.1) results in $\sum_{i=1}^{\infty} x_1^i = 1$. For $0 \leq x_1 < 1$ the left-hand-side of the equation is the geometric series, and the sum converges to $\sum_{i=1}^{\infty} x_1^i = \frac{1}{1-x_1} - 1$. By substituting in (4.1.1) the equation simplifies to $\frac{1}{1-x_1} - 1 = 1$, with solution $x_1 = \frac{1}{2}$. Thus, the achievable competitive ratio is $\frac{1}{x_1} = 2$, which is the result of the classical ski rental problem.

For the intermediate values of $k$, we get monotone decreasing real roots of $x_1$ in the interval $[0.5, \frac{1}{\Phi}]$ as shown in Figure 4.1.1 for $k \in \{2, \cdots, 20\}$ and thus monotonically increasing competitive ratios $C(k)$ for generalized $k$-day ski rental problem from $\Phi$ to 2.

**Lemma 4.1.** Let $C(k) = \frac{1}{x_1}$ be the reciprocal of the single real root $\tilde{x}_1$ of the equation $\sum_{i=1}^{k} x_1^i = 1$ in the interval $[0, 1]$. The best achievable competitive ratio for the generalized $k$-day ski rental problem is the value $C(k)$. 
Chapter 4. Renting skis and managing delays online

Figure 4.1.1: The root values of $\sum_{i=1}^{k} x_i = 1$ in the interval $[0, 1]$ for several values of $k$. The root value of $x_1$ rapidly converges to $\frac{1}{2}$.

Thus, the knowledge that the game between the skier and the adversary lasts for at most $k$ days gives a possible gain of $2 - C(k)$ for the competitive ratio. In fact, the following modification of the classical ski rental algorithm attains this competitive ratio.

In step $i < k$ of the game the skier buys the skis if $\frac{x_{i-1}+1}{x_i} \leq C(k)$. If the skier reaches step $k$ she deterministically buys the skis.

The above analysis shows that if step $k$ is reached the competitive ratio must be smaller than $C(k)$ (since otherwise the set of $x_i$ values would be a witness for a better adversarial strategy, in contradiction to the above lemma). We call this simple algorithm the $C(k)$-pondering algorithm.

Lemma 4.2. The $C(k)$-pondering algorithm achieves a competitive ratio of $C(k)$ for the generalized $k$-day ski rental problem.

4.2 Online delay management

In this section, we specify the model we are analyzing for delay management, and explain which information is disclosed online. Next, we introduce some notation which we use in our competitive analysis. Then, we show the relation between one restricted setting of the
delay management problem and the discounted ski rental problem.

4.2.1 Model, notation and online information disclosure

The basic setting for our model is as described in Section 2.2, with additional restrictions addressing several key aspects. First, we generally restrict the railway network to have the topology of a simple path, a railway corridor. Then, we generally consider two options to serve the legs of the railway network: either they are all served by a single train, which thus performs intermediate stops, or each train serves exactly one leg. Thus, we consider restricted settings of an online version of the binary delay management problem and of the binary delay management problem with intermediate stops having restricted train services. In some part of our analysis, we assume, as we have also done in some of our offline variants, that all passenger paths travel to the last node of the railway corridor. Unless otherwise stated, we want to minimize the total passenger delay.

Online information disclosure We remain with specifying which part of the instance is disclosed in an online fashion and which decisions shall be taken online. We assume the complete delay management instance to be given as input, with the exception of the source delays. Thus, an online algorithm is given the railway network $G$, the set of passenger paths $\mathcal{P}$, the weight of the passenger paths $w$, the period $T$, the set of train services $\mathcal{R}$ at the beginning. The source delays of the passenger paths, however, are disclosed in an online fashion, in a sequence according to the implicit time information given by the nodes of the railway network. Specifically, when considering the next node of the sequence with respect to the time information, the adversary discloses the source delays of all passenger paths which have the considered node as origin node. The online algorithm must, by considering this information, and all previously disclosed information and the already taken decisions, reply by stating which outbound legs from the considered node wait for the connecting passengers, and which outbound legs depart as scheduled. Moreover, the algorithm may not revoke any previous decisions. Note that the considered sequence of online information disclosure reflects the intuitive idea that we know which passengers wishing to board a train are late when
each the train is about to leave, and an operator must take the decisions on which trains wait in a sequence according to the departure time of the trains.

In general, one might know that passenger wishing to connect to a train are delayed long before the departure time of that train (think about delayed long distance trains). However, we do not consider this possibility, as in an online setting, which consider worst-case scenarios, it is most advantageous for the adversary to disclose as little information as possible. Disclosing the source delays of passenger paths before the point in time when the online algorithm is bound to decide on the first leg boarded by the considered passenger path is only advantageous for the online algorithm, and this possibility can thus be neglected.

We point out that in the setting on a railway corridor, the sequence of information disclosure and decision by the online algorithm corresponds to the following pattern: first, the adversary discloses the source delays of the passenger paths boarding the first leg of the corridor; the online algorithm replies with the delay policy for the first leg. Then, the adversary proceeds with disclosing the source delays of the passenger paths boarding the subsequent leg of the corridor, and the online algorithm declaring the delay policy for that leg, and so on. Thus, the explicit time information of nodes and legs is irrelevant also in this setting.

**Notation** In Section 4.3.1, we consider the simple case of a railway corridor \( G = (V, E) \) which is served by a single train. Therefore, we define \( V = \{1, \cdots , n\} \), and \( E = \{(i, i + 1) | i \in \{1, \cdots , n - 1\}\} \). Thus, the train serving the network sequentially serves the nodes in the order \( 1, \cdots , n \) and, as a consequence, the train service is specified by \( r = \{(1, 2), \cdots , (n - 1, n)\} \). In this case, if train service decides to wait at some node \( i \), then all legs \( (j, j + 1), j \in \{i, \cdots n - 1\} \) of the corridor do also wait, since a train service cannot catch up on delays on any of its legs. Thus, the only decision that needs to be taken in this setting is beginning from what node the legs of the train service wait. One possibility, of course, is that the outbound leg of no node at all waits, meaning that it always departs (and arrives) on time. We refer to this model as **online delay management for a single train on a railway corridor**.

For the analysis of this setting, we require some additional nota-
4.2. Online delay management

tion, which is discussed next. First, we consider the knowledge of the source delays from an offline point of view, that is, for a known and fixed configuration of the source delays. For this case, the variables $O \geq i$ and $D^i$ below aggregate the total weight of the passenger paths at node $i$ that are affected by the decision of the outbound leg to either wait or to depart on time.

$O \geq i$: The total weight of all passenger paths that are source punctual, and either (i) have a origin node before node $i$, and a destination node after node $i$, or (ii) have a origin node at or after node $i$. This is total weight of the passenger paths that are affected if the train service begins to wait with the leg outbound from node $i$.

$D^i$: The total weight of all passenger paths that are source delayed and have node $i$ as origin node. This is the total weight of the passenger paths that are dropped at node $i$ if the outbound leg from $i$ departs on time.

Further, we denote by $\Delta(j)$ the value of the objective function if the outbound leg from node $j$ waits (and as a consequence, all subsequent legs). Since our objective function is the total passenger delay, it is defined as

$$\Delta(j) = (T - \delta) \sum_{i<j} D^i + \delta \sum_{i=1}^{n} D^i + \delta O \geq j,$$

which we wish to minimize over all nodes $j \in V$. Notice that the total weighted source delay $\delta \sum_{i=1}^{n} D^i$, which cannot be optimized, is included in the objective function $\Delta(j)$. Section 4.5.2 shows that, in some sense, no online algorithm can have a bounded competitive ratio if this part is not included in the objective function.

We further define analogous total passenger weight variables $o \geq i$ and $d^i$ for the point of view of an online algorithm, using the same letters in lowercase:

$o \geq i$: The total weight of all passenger paths that either (i) are source punctual, have origin node before or at node $i$, and a destination node after node $i$, or (ii) have a origin node after node $i$, regardless of their source delay. This weight reflects an online algorithm’s worst-case estimate about the current and future source punctual passenger paths, and is therefore an upper
Chapter 4. Renting skis and managing delays online

bound on the overall weight of the passenger paths that will be affected if the outbound leg from node \( i \) waits.

\( d^i \): The total weight of all passenger paths with a source delay and node \( i \) as origin node. This is the total weight of the passenger paths that will be dropped at node \( i \) if the outbound leg from node \( i \) departs on time.

Due to their online character, these variables only have a meaning for all nodes \( j \leq i \) up to the last node \( i \) where the adversary has disclosed the source delays. In particular, when the source delays for the passenger paths with origin in node \( i \) are revealed, an online algorithm knows the correct total weight of all source delayed passenger paths with origin node up to \( i \): \( d^j = D^j, \forall j \leq i \).

Further, the following relations hold:

\begin{align*}
O_{\geq i+1}^i &\leq O_{\geq i}^i & (4.2.1) \\
o_{\geq i+1}^i &\leq o_{\geq i}^i & (4.2.2) \\
O_{\geq i}^i &\leq o_{\geq i}^i & (4.2.3) \\
o_{\geq i}^i &= O_{\geq i}^i + \sum_{j > i} D^j \leq O_{\geq i}^i + \sum_{j} D^j & (4.2.4)
\end{align*}

Inequalities (4.2.1) and (4.2.2) hold because at each node passenger paths may alight from the leg. Hence, the total weight of the passenger paths that are influenced by a decision to wait monotonically decreases if the train service starts to wait at a later node. Inequality (4.2.3) holds, as the online algorithm does not know which passenger paths having origin node after \( i \) have a source delay. Hence, \( o_{\geq i}^i \) is an upper bound on the overall weight of the passenger paths that are delayed by the decision to start waiting at node \( i \). Inequality (4.2.4) states that this overestimate equals the actual overall weight of the passenger paths influenced by this decision (i.e. with the offline knowledge on the source delays), plus the total weight of all source delayed passenger paths with origin node after node \( i \). This weight can naturally be bounded as shown in inequality (4.2.4).

4.2.2 Delay management and ski rental

In the following, we present a one-to-one correspondence between the discounted ski rental problem and a restricted version of the online
4.2. Online delay management

delay management problem for a single train on a corridor. Here, we restrict all passenger paths to have node $n$ as destination. Within the objective $\Delta(j)$, consider the terms $\delta \sum_i D^i + \delta O^{\geq j}$, that is, the weighted delay of the passenger paths with arrival delay $\delta$, plus the weighted delay $\delta$ of all dropped passenger paths. When all passengers have node $n$ as destination, and given that one leg eventually waits for delayed passenger paths, the value of these terms is constant and thus independent of the decision taken by an algorithm, since all passenger paths have an arrival delay of at least $\delta$.

We map this constant sum to the original undiscounted price of buying the skis in the discounted ski rental problem. Further, the total weighted arrival delay of the dropped passenger paths at node $j$ corresponds to the renting price of the skis on day $j$. So, there is a bijection between waiting at the outbound leg from node $j$ and buying the skis on day $j$. With this idea, the cost of buying the skis on day $j$ corresponds to the delay caused by waiting at node $j$, whereas the cost of renting the skis on day $j$ corresponds to the cost of dropping the source delayed passenger paths with origin at node $j$. By setting the discount factor for buying the skis to $\alpha = \frac{\delta}{T}$, we complete the mapping. Indeed, the cost of starting to wait at node $j$ is the same as the cost of buying the skis on day $j$:

$$\text{ski-cost}(j) = T \sum_{i < j} D^i + \delta \sum_i D^i + \delta O^{\geq j} - \frac{\delta}{T} T \sum_{i < j} D^i = \Delta(j)$$

In the next section, we show that a strategy similar to classical one for the ski rental problem is 2-competitive for the online delay management problem. Hence, the above mapping provides some intuition for the structure of our online algorithm for online delay management for a single train on a corridor. Moreover, its analysis also applies to the above discounted ski rental problem.

We point out that the more general delay management setting with a single train on a corridor is structurally different from both the discounted ski rental problem and the classical ski rental problem. The fundamental difference lies in the fact that passenger paths may end before the last node of the corridor. In fact, in both ski rental problems the key decision from an offline point of view is whether or not to buy skis. Hence, the decision is boolean. In contrast, an optimal offline solution to the delay management problem must not only decide whether to wait or not, but additionally at which node it should
start to wait. Indeed, it can be advantageous for an offline algorithm to start by dropping some source delayed passenger paths with origin near to the beginning of the corridor, and start to wait later on. Indeed, this strategy would allow to bring passenger paths of high weight at their destination with zero arrival delay, and maintain connections to the remaining source delayed passenger paths. Finally, the number of nodes on the corridor is known a priori in the online delay management problem. Thus, differently from the classical ski rental problem, it is known how many times a decision has to be taken. We have already shown that the finiteness of the problem is a crucial difference also in some settings of the ski rental problem.

4.3 Competitive online algorithms

This section presents several online algorithms for the special cases of the online delay management problem specified in Section 4.2.1. The presented online algorithms are inspired by the 2-competitive algorithm for the classical ski rental problem. With this relation as background, Section 4.3.1 describes a family of 2-competitive online algorithms for the case in which a single train serves a railway corridor. Subsequently, Section 4.3.2 shows that this family of algorithms can be extended to a 2-competitive algorithm for the more complex setting on a corridor where each train serves exactly one leg. Finally, we show in Section 4.3.3 that the latter algorithm also works for railway networks with an in-tree topology, when all passenger paths travel to the same destination node, using the same final leg.

4.3.1 A family of 2-competitive algorithms for a single train on a corridor

This section presents a family of 2-competitive online algorithms for the online delay management problem for a single train on a railway corridor, as described in Section 4.2.1. The online algorithms resemble the classical online algorithm for the ski rental problem. Loosely speaking, the single train service should start waiting at the outbound leg from node \( j \) if the weighted delay caused by dropping passenger paths with origin node up to and including node \( j \) exceeds the weighted delay caused to a worst-case estimate of current and future
4.3. Competitive online algorithms

on time passenger paths by waiting in node $j$.

As we present a family of algorithms, we must be a little more precise. For any value $t \in [T - \delta, T]$, the online algorithm $\text{ALG}(t)$ of the family starts waiting at the outbound leg from node $j$ if (using the notation from Section 4.2.1)

$$t \sum_{i \leq j} d_i \geq \delta o^{\geq j}.$$ 

Note that the two extremal values of $t$ lead to two extremal behaviors within the family of algorithms. The online algorithm $\text{ALG}(T - \delta)$ starts to wait as late as possible. In contrast, the online algorithm $\text{ALG}(T)$ starts to wait as early as possible. Below, we show that both extremal algorithms are 2-competitive.

Intuitively, the online algorithms $\text{ALG}(t)$ achieve the competitive ratio of 2 by a similar argument as for the ski rental problem: they drop all source delayed passenger paths, up until the node $j$ where the accumulated total passenger delay balances a worst case estimate on the total passenger delay which would be caused by starting to wait. At this point, an adversary can set all future passenger paths with origin node greater than $j$ to be source punctual. This adversarial strategy causes the same amount of weighted delay to the online algorithm as it has already accumulated, whereas the optimal decision would have been not to wait anywhere.

**Theorem 4.3.** The family of online algorithms $\text{ALG}(t)$, for values of $t \in [T - \delta, T]$, that start to wait at the outbound leg from node $j$ if

$$\left(t \sum_{i \leq j} d_i\right) \geq \delta o^{\geq j},$$

is 2-competitive for the online delay management problem for a single train on a railway corridor.

**Proof.** Let $j^*$ be the node where an optimal offline solution makes the outbound leg wait, and $j$ the node where the online algorithm $\text{ALG}(t)$ makes the outbound leg wait. In fact, unless it is optimal, the online algorithm $\text{ALG}(t)$ either starts waiting too late at $j > j^*$, or it started waiting too early at $j < j^*$. The proof distinguishes these two cases.

**Case $j > j^*$.** In this case, $\text{ALG}(t)$ starts waiting too late. For the worst-case analysis, we consider the algorithm $\text{ALG}(T - \delta)$ that within the family starts waiting the latest. The objective value for
\[ \Delta(j) = (T - \delta) \sum_{i < j} D^i + \delta \sum_{i} D^i + \delta O_{\geq j} \]

\[ = (T - \delta) \sum_{i < j^*} D^i + \delta \sum_{i} D^i + \delta O_{\geq j^*} \]

\[ \underbrace{\Delta(j^*)} \]

\[ - \delta O_{\geq j^*} + \delta O_{\geq j} + (T - \delta) \sum_{i = j^*}^{j-1} D^i, \quad (4.3.1) \]

where \( \Delta(j^*) \) is the objective value of the optimal offline solution. Since \( j > j^* \), inequality (4.2.1) implies that \(-\delta O_{\geq j^*} + \delta O_{\geq j} \leq 0\). Further,

\[ (T - \delta) \sum_{i = j^*}^{j-1} D^i \leq (T - \delta) \sum_{i \leq j-1} D^i. \]

Since \( \text{ALG}(T - \delta) \) decided not to make the outbound leg from node \( j - 1 \) wait, it holds that

\[ (T - \delta) \sum_{i \leq j-1} D^i \leq \delta O_{\geq j-1} \leq \delta \left( O_{\geq j-1} + \sum_{i} D^i \right) \]

\[ \leq \delta \left( O_{\geq j^*} + \sum_{i} D^i \right) \leq \Delta(j^*), \]

where the second inequality follows from inequality (4.2.4). Putting everything together, we see that equation (4.3.1) can be bounded as

\[ \Delta(j) \leq \Delta(j^*) - \delta O_{\geq j^*} + \delta O_{\geq j} + \delta \left( O_{\geq j^*} + \sum_{i} D^i \right) \leq 2\Delta(j^*). \]

**Case \( j < j^* \).** Similar to the previous case, we consider the worst case online algorithm \( \text{ALG}(T) \) that starts waiting earliest, which has objective value

\[ \Delta(j) = \Delta(j^*) - \delta O_{\geq j^*} + \delta O_{\geq j} - (T - \delta) \sum_{i = j}^{j^*-1} D^i. \]
Inequality (4.2.3) and the fact that ALG(T) starts waiting at the out-bound leg from node \( j \) imply
\[
\delta O \geq \delta o \geq \mathcal{J} \leq T \sum_{i \leq j} D^i \leq T \sum_{i \leq j^*} D^i.
\]
Since
\[
T \sum_{i \leq j^*} D_i \leq (T - \delta) \sum_{i \leq j^*} D_i + \delta \sum_{i \leq j^*} D_i \leq \Delta(j^*),
\]
we finally have
\[
\Delta(j) \leq \Delta(j^*) - \delta O \geq \mathcal{J} + \Delta(j^*) - (T - \delta) \sum_{i = j}^{j^*-1} D^i \leq 2\Delta(j^*).
\]

We have thus shown that the whole family of online algorithms \( \text{ALG}(t), \ t \in [T - \delta, T] \) is 2-competitive. The following corollary shows that the above analysis is in fact tight.

**Corollary 4.4.** The competitive analysis of the family of online algorithms \( \text{ALG}(t), \ t \in [T - \delta, T] \) for the online delay management problem for a single train on a railway corridor is tight.

**Proof.** We show that the analysis is tight for both cases in the proof of Theorem 4.3, again considering the worst case algorithms \( \text{ALG}(t) \) for each case.

**Case** \( j > j^* \). Consider a corridor consisting of the sequence of three nodes \( V = \{1, 2, 3\} \). Two passenger paths \( P_0 = \{1, 2\} \) and \( P_1 = \{2, 3\} \) wish to travel on the corridor, with weights \( w(P_0) \) and \( w(P_1) \), respectively, where \( w(P_1) = \frac{T - \delta}{\delta} w(P_0) + \epsilon \).

As usual, the online algorithm, \( \text{ALG}(T - \delta) \) in this case, does not know a-priori whether these passenger paths are source delayed or not. At node 1, the adversary declares \( P_0 \) to be source delayed. As \( (T - \delta)w(P_0) < \delta w(P_1) = (T - \delta)w(P_0) + \delta \epsilon \), \( \text{ALG}(T - \delta) \) will not make the outbound leg from node 1 wait. Upon the arrival of the leg at node 2, the adversary also declares \( P_1 \) to be source delayed. Then, \( \text{ALG}(T - \delta) \) will certainly wait in node 2, as this will not delay any other passenger paths. The optimal offline solution makes the
outbound leg from node 1 wait (and thus also all consecutive legs wait), so $\text{ALG}(T-\delta)$ indeed started to wait too late. The ratio between the online solution value $\Delta(2)$ and the optimal offline solution value $\Delta(1)$ is

$$\frac{\Delta(2)}{\Delta(1)} = \frac{Tw(P_0) + \delta w(P_1)}{\delta \left(Tw(P_0) + w(P_1)\right)} = 2 \frac{Tw(P_0) + \delta \epsilon}{Tw(P_0) + \delta \epsilon} - \frac{\delta w(P_0) + \delta \epsilon}{Tw(P_0) + \delta \epsilon} = 2 - \frac{\delta w(P_0) + \delta \epsilon}{Tw(P_0) + \delta \epsilon}.$$

For $\epsilon \to 0$ the ratio converges to $r = 2 - \frac{\delta}{T}$.

**Case $j < j^\ast$.** Consider the corridor $V = \{1, 2, 3\}$ with passenger paths $P_0 = \{1, 2, 3\}$, $P_1 = \{1, 2\}$, and $P_2 = \{2, 3\}$, with weights $w(P_0)$, $w(P_1)$ and $w(P_2) = \frac{T}{\delta} w(P_0) - \epsilon$. At node 0, the adversary reveals $P_0$ to be source delayed and $P_1$ to be source punctual. The worst case algorithm $\text{ALG}(T)$ for this case does not wait at node 0 since $\delta(w(P_1) + w(P_2)) > Tw(P_0)$ for $\delta w(P_1) > \delta \epsilon$. When the leg arrives at node 2, the adversary discloses $P_2$ to be source punctual as well. But then, $Tw(P_0) > \delta w(P_2) = Tw(P_0) - \delta \epsilon$, so $\text{ALG}(T)$ waits at node 2, interestingly enough without anybody to wait for. The optimal offline solution does not wait anywhere, denoted by the symbol $\emptyset$, thus dropping only the passenger path $P_0$, and showing that $\text{ALG}(T)$ started waiting too early. In this case, the competitive ratio between the online solution value $\Delta(2)$ and the optimal offline solution value $\Delta(\emptyset)$

$$\frac{\Delta(2)}{\Delta(\emptyset)} = \frac{Tw(P_0) + \delta w(P_2)}{Tw(P_0)} = \frac{2Tw(P_0) - \delta \epsilon}{Tw(P_0)} \epsilon \to 0 = 2.$$

This ratio is independent from the parameters $\delta$ and $T$, and the analysis directly shows its tightness.

For the case $j > j^\ast$, the analysis is tight up to $r = 2 - \frac{\delta}{T}$. For $\epsilon' = \frac{\delta}{T}$ and letting $\epsilon' \to 0$, $r$ gets arbitrarily close to 2, so that analysis is tight as well.

### 4.3.2 From a single train to multiple trains on a corridor

This section studies the less restricted variant of a railway corridor with nodes $V = \{1, \ldots, n\}$, where each train serves exactly one leg,
4.3. Competitive online algorithms

Figure 4.3.1: The network forming a corridor with all passenger paths traveling to the same destination. Arrows indicate legs, lines the passenger paths.

shown in Figure 4.3.1. So, each train uniquely corresponds to an edge, no train service has intermediate stops, and each leg, except for the first and the last, serves as a connection for exactly one leg and has a connection to exactly one (consecutive) leg. Furthermore, we assume that all passenger paths have node $n$ as their destination node. By setting this focus, each passenger path is characterized by its origin node. We refer to this setting as online delay management for multiple trains on a terminal railway corridor.

The following observation considers the structure of optimal delay policies for the setting described above.

**Observation 4.5.** For online delay management for multiple trains on a terminal railway corridor, there exists an optimal delay policy which does not drop any source punctual passenger path, thus resulting in a sequence of non-waiting legs followed by a sequence of waiting legs.

*Proof.* Consider any delay policy $\Pi$ which drops some source punctual passenger paths, and focus on its last wait/non-wait transition, that is, the last two consecutive legs $e_i, e_{i+1}$ where the leg $e_i$ waits and leg $e_{i+1}$ departs as scheduled. So, this last transition occurs at node $i + 1$, and no passenger path with origin node $j < i$ will be maintained at node $i + 1$. We conclude that the policy $\Pi$ drops all passenger paths with source node $j < i$, regardless of their source delay.

In contrast, consider the delay policy $\Pi'$ that makes the consecutive legs $e_1, \ldots, e_i$ depart on time, and that takes the same decisions as $\Pi$ for the legs $e_{i+1}, \cdots, e_{n-1}$. Up until node $i$, the policy $\Pi'$ only drops source delayed passenger paths with origin nodes $j < i$, which results in less dropped passenger paths than in the policy $\Pi$. Thus, we have a smaller weighted arrival delay for the passenger paths with origin nodes $j < i$. As $\Pi'$ behaves exactly as $\Pi$ after node $i + 1$, it
follows that $\Pi'$’s objective value is at most that of $\Pi$. Moreover, the policy $\Pi'$ drops no source punctual passenger paths. 

The structure of the delay policy resulting from this observation is equivalent to the structure of any solution from our family of $\text{ALG}(t)$ algorithms for the single train on a corridor. Therefore, this family of online algorithms is also 2-competitive for multiple trains on a terminal corridor.

**Observation 4.6.** The family of online algorithms $\text{ALG}(t)$, $t \in [T - \delta, T]$, for online delay management for a single train on a railway corridor are also 2-competitive for online delay management for multiple trains on a terminal railway corridor.

**Proof.** We interpret each leg of the railway corridor served by the single train a leg served by a single train in the current setting, and apply the online algorithm $\text{ALG}(t)$. By definition, $\text{ALG}(t)$ constructs a solution consisting of a sequence of non-waiting legs, followed by a sequence of waiting legs, and Theorem 4.3 states that this solution is at most a factor 2 away from the optimal offline solution. This optimal offline solution has at most one non-wait/wait transition, and Observation 4.5 states that an optimal solution with the same structure exists for multiple trains on a terminal corridor. So, the claim follows.

Although this already gives a family of 2-competitive algorithms for multiple trains on a terminal corridor, we next describe another, slightly different 2-competitive online algorithm. We do so because this algorithm has a very simple structure, is directly linked to the ski rental analogy from Section 4.2.2, and, most importantly, can be generalized to railway networks with a more complex topology.

Inspired by Observation 4.5, the online algorithm constructs a solution that consists of a sequence of non-waiting legs $e_1, \ldots, e_{i-1}$, followed by a sequence of waiting legs $e_i, \ldots, e_{n-1}$. For determining the node $i$ at which the first leg of the sequence waits, the algorithm compares two weights at each node $i$: the so far accumulated weighted cost of dropping source punctual passenger paths, and the parameter $\delta_{\text{max}} = w \cdot \delta$, where $w = \sum_{p \in P} w(P)$ is the sum of the weights of the passenger paths in the instance. Note that, since $w$ is the total weight of all source delayed and source punctual passenger
paths, it is known to the online algorithm. Whenever the accumulated dropping costs exceed $\delta_{\text{max}}$, the algorithm starts the sequence of waiting legs.

In order to prove the competitive ratio of this algorithm, we need another observation on the structure of an optimal delay policy. To that end, let $w_\delta$ be the total weight of all source delayed passenger paths from an offline point of view.

**Observation 4.7.** Any optimal solution for multiple trains on a terminal corridor either drops all source delayed passenger paths, or drops no passenger paths at all.

**Proof.** By Observation 4.5, consider any solution that does not drop any source punctual passenger path and is composed of a sequence of non-waiting legs followed by a sequence of waiting legs. If a solution starts the sequence of waiting legs after having dropped an amount $k$ of the total passenger weight $w$, it has objective value $k \cdot (T - \delta) + w \cdot \delta$. This expression achieves its minimum $\delta_{\text{max}} = w \cdot \delta$ at $k = 0$, meaning that no source delayed passenger paths are dropped either. On the other hand the solution may drop all source delayed passenger paths, resulting in an objective value $w_\delta \cdot T$. Hence, the optimal solution is defined by the minimum of $\{\delta_{\text{max}}, w_\delta \cdot T\}$, and correspondingly either drops all source delayed passenger paths, or no passenger path at all.

**Theorem 4.8.** The online algorithm SIMPLE($\delta_{\text{max}}$) that makes the sequence of waiting legs start with the outbound leg from node $j$ if $T \sum_{i \leq j} d^i \geq \delta_{\text{max}}$, is 2-competitive for online delay management for multiple trains on a terminal railway corridor.

**Proof.** If the optimal offline solution is to drop all source delayed passenger paths, then $\delta_{\text{max}} > w_\delta \cdot T$. Clearly, $w_\delta \cdot T \geq \sum_{i \leq j} T d^i$ for all nodes $j \in V$, meaning that SIMPLE($\delta_{\text{max}}$) does not make any leg wait, and in fact constructs the optimal offline solution.

On the other hand, suppose the optimal solution is to drop no passenger paths at all, meaning that it has objective value $\delta_{\text{max}} < w_\delta \cdot T$. In this case, SIMPLE($\delta_{\text{max}}$) starts its sequence of waiting legs at some node $j$, giving costs $\sum_{i < j} T d^i \leq \delta_{\text{max}}$ for dropping source delayed passenger paths with origin node up to node $j$. Additionally, it has costs at most $\delta_{\text{max}}$ for the arrival delay of the remaining passenger paths, yielding a total objective value of at most $2\delta_{\text{max}}$. 

\[ \square \]
Chapter 4. Renting skis and managing delays online

Basically, the algorithm SIMPLE($\delta_{\text{max}}$) achieves its competitive ratio of 2 by exploiting the fact that all passenger paths have a common destination node, and the resulting structure of an optimal solution. In the next section, we argue that the same reasoning applies to more general network topologies.

4.3.3 From a corridor to tree-like networks with a common terminal leg

Theorem 4.8 and the online algorithm SIMPLE($\delta_{\text{max}}$) can be extended to the case where multiple trains serve a more complex railway network, with all passenger paths still having as destination one common terminal node.

To that end, first consider a network with the topology of an in-tree, that is, $G$ is a tree with all edges directed towards a root node $n$. We assume that all passenger paths share the last edge $e_{n-1}$ to the root node $n$. Note that any in-tree not satisfying this condition can be subdivided into a forest of disjoint in-trees, each satisfying the condition. We construct a separate in-tree for each inbound leg to the terminal node, and copy the terminal node for each branch. Since the resulting forest consists of disjoint and independent in-trees, it is equivalent to the original in-tree. Any online algorithm for a single in-tree can be applied to the forest while maintaining the competitive ratio. Therefore, the assumption on the common last edge is without loss of generality.

For simplicity but without loss of generality, assume the trains services have no intermediate stops. Thus, each leg is served by exactly one train, and each train serves exactly one leg. The rest of the problem follows the previously stated definitions.

We consider an online setting with a global view, that is, any online algorithm knows the network structure, the passenger paths, and their weights. The source delays of the passenger paths are sequentially disclosed in a time consistent fashion: whenever a passenger path with origin node $i$ is disclosed as source delayed or source punctual, all passenger paths boarding a leg earlier than node $i$’s time have already been disclosed. Hence, the online algorithm maintains a cut-like time front on the known source delays which grows from the leaves of the in-tree towards the root node, see Figure 4.3.2. We represent the development of this time front by an ordering on the node set...
4.3. Competitive online algorithms

Figure 4.3.2: The in-tree railway network with all passenger paths having as destination the root node of the in-tree. Solid lines represent two explanatory passenger paths. The dashed line represents the cut-like front showing the disclosed information by the adversary.

$V$, and for convenience assume the ordering to be $V = \{1, \ldots, n\}$, with $n$ the root node of the in-tree.

**Corollary 4.9.** The online algorithm SIMPLE($\delta_{\text{max}}$) of Theorem 4.8, adapted to the time ordering $(1, \ldots, n)$ of the nodes, is 2-competitive for a railway network with an in-tree topology with multiple train services and all passenger paths using the common terminal leg $e_{n-1}$.

**Proof.** Similar to the previous proofs, there exists an optimal solution which maintains all source punctual passenger paths, since dropping a source punctual passenger path is never beneficial. Hence, on each leaf-to-root path of the in-tree, the optimal solution has the structure of the optimal solution for online delay management for multiple trains on a terminal railway corridor. Again, a decision to start waiting at any node delays the total passenger weight by $\delta$, since it implies that the terminal leg $e_{n-1}$ waits. Thus, the optimal solution either never waits and drops all source delayed passenger paths, or waits for all source delayed passenger paths, thus dropping no passenger paths at all. With this insight, the analysis follows the same lines as for Theorem 4.13.

In fact, one can see that the same reasoning applies to any railway network where all passenger paths use a common terminal leg. Again, we assume an ordering $(1, \ldots, n)$ of the nodes that is time consistent with the disclosure of the passenger path delays.
Corollary 4.10. The online algorithm $\text{SIMPLE}(\delta_{\max})$ of Theorem 4.8, adapted to the time ordering $(1, \ldots, n)$ of the nodes, is 2-competitive on any railway network topology where all passenger paths travel to the terminal node $n$ using a common terminal leg $e_{n-1}$.

Proof. As before, dropping source punctual passenger paths is never beneficial. Thus, any decision to start waiting at some node results in a delay of size $\delta$ for the total passenger weight, and the trade-off for the optimal solution is still between dropping all source delayed passenger paths and not dropping any passenger paths. \hfill $\square$

4.4 Lower bounds on the competitive ratio

This section presents two lower bounds on the competitive ratio for the online delay management problem on railway corridors. We first prove a Golden ratio lower bound for online delay management for a single train on a railway corridor, using a mathematical programming approach. Next, we derive a lower bound of 2 using a ski rental-like argumentation for the online delay management problem for multiple trains on a terminal railway corridor.

4.4.1 Golden ratio competitiveness

Below, we prove that no online algorithm can have a competitive ratio below $\Phi$ for the online delay management problem for a single train on a railway corridor (as defined in Section 4.2.1), where $\Phi = \frac{\sqrt{5}+1}{2}$ is the Golden ratio.

Theorem 4.11. No online algorithm can have a competitive ratio better than $\Phi$ for the online delay management problem for a single train on a railway corridor.

Proof. The proof is based on the network of Figure 4.4.1. The instance has three passenger paths. The passenger path $P_1$ has weight $w(P_1)$, origin node in $A$ and destination node $C$; $P_2$ has weight $w(P_2)$, origin node $A$ and destination node $C$; $P_3$ has weight $w(P_3)$, origin node $B$ and destination in node $C$. The legs defining the passenger paths are clear, since we are on a railway corridor. Finally, at
4.4. Lower bounds on the competitive ratio

![Diagram](image-url)

**Figure 4.4.1:** The simple network used in the proof of Theorem 4.11. Passenger path \( P_1 \) is source punctual, Passenger path \( P_2 \) is source delayed, whereas the source delay of \( P_3 \) is yet to be disclosed by the adversary.

The beginning the adversary declares \( P_1 \) to be source delayed and \( P_2 \) to be source punctual.

In this situation, any online algorithm must decide whether to make the outbound leg from node \( A \) wait or not. The adversary enforces that, whatever decision is taken, it is \( c \)-competitive. Thus, he chooses the parameters \( \delta, T, w(P_1), w(P_2), w(P_3) \) such that \( c \) is maximal. This is described by the following mathematical program:

\[
\begin{align*}
\max c \\
\delta(w(P_1) + w(P_2) + w(P_3)) & \geq cT w(P_1) \quad (4.4.1) \\
Tw(P_1) + \delta(w(P_2) + w(P_3)) & \geq c\delta(w(P_1) + w(P_2) + w(P_3)) \quad (4.4.2) \\
T(w(P_1) + w(P_3)) & \geq c\delta(w(P_1) + w(P_2) + w(P_3)) \quad (4.4.3)
\end{align*}
\]

Inequality (4.4.1) reflects the situation where the online algorithm makes the outbound leg from \( A \) wait, thus delaying all three passenger paths, but it would have been better not to wait at all because \( P_3 \) is source punctual. The left hand side (lhs) reflects the costs of the online algorithm, the right hand side (rhs) the costs of the optimal offline solution, weighted with the competitive ratio \( c \). Inequality (4.4.2) reflects the situation where the online algorithm makes the outbound leg from \( A \) depart as scheduled but makes the outbound leg from \( B \) wait, the adversary declares a source delay on \( P_3 \), but where it would have been better to make the outbound leg from \( A \) wait. Finally, inequality (4.4.3) describes the situation where the online algorithm decides not to make any leg wait, even if \( P_3 \) has a source delay, but it would have
been better to make all legs wait. For this last online strategy we do not consider the case where the optimal offline delay policy makes only the outbound leg from $B$ wait. If this strategy were better than making the outbound leg from $A$ wait, it would only make the bound on the competitive ratio bigger than shown here.

For simplicity, we normalize all weights of the passenger paths with respect to $w(P_1)$, and the delays with respect to $\delta$. Hence, $P_1$ has weight 1, and $\delta = 1$. The mathematical program becomes:

$$\max c$$

$$1 + w(P_2) + w(P_3) \geq cT$$ (4.4.4)

$$T + w(P_2) + w(P_3) \geq c(1 + w(P_2) + w(P_3))$$ (4.4.5)

$$T(1 + w(P_3)) \geq c(1 + w(P_2) + w(P_3))$$ (4.4.6)

We restrict our attention to the case $w(P_2) \leq (T - 1)w(P_3)$. In this case, (4.4.5) is tighter than (4.4.6), so we can omit the latter equation. As we are constructing a specific solution to the mathematical program, we let inequality (4.4.4) be tight. Note that by choosing (4.4.5) to be tight, we can construct an example with the same competitive ratio as shown below. Now we can set $w(P_3) = cT - 1 - w(P_2)$. Thus, substituting into (4.4.5):

$$T + w(P_2) + cT - 1 - w(P_2) \geq c + cw(P_2) + c^2T - c - cw(P_2)$$

$$T(1 + c) - 1 \geq c^2T$$

Let $c = \Phi - \epsilon$, and recall that $\Phi + 1 = \Phi^2$:

$$T(1 + \Phi) - T \epsilon - 1 \geq \Phi^2T - 2\Phi \epsilon T + \epsilon^2T$$

$$(2\Phi - 1 - \epsilon)\epsilon T \geq 1$$ (4.4.7)

As long as $(2\Phi - 1 - \epsilon)\epsilon \geq 0$, we can choose $T$ such that (4.4.7) is satisfied. The condition is satisfied for $0 < \epsilon \leq 2\Phi - 1$, and we can set $T = \frac{1}{(2\Phi - 1 - \epsilon)\epsilon}$. By letting $\epsilon \to 0$, $c$ becomes arbitrarily close to $\Phi$. This shows that the competitive ratio of any online algorithm cannot be better than $\Phi$.

A closer inspection of the constructed instance shows that, within the setting of this example, we cannot prove the competitive ratio to
4.4. Lower bounds on the competitive ratio

be greater than \( \Phi \), as choosing \( \epsilon \) to be negative leads to a contradiction. \( \square \)

### 4.4.2 Ski-rental-like lower bounds

For the online delay management problem with multiple trains on a terminal railway corridor (see Section 4.3.2), we derive a ski rental-like lower bound of 2. First, we observe that any deterministic algorithm is basically characterized by a single number that states the sum of the passenger path’s weights the algorithm drops before starting to wait. Note that this is similar to the ski rental problem, where this number specifies the total accumulated renting cost.

**Observation 4.12.** Any deterministic online algorithm for the setting of a single train on a corridor can be parameterized by the number \( s := p' \cdot \frac{T - \delta}{\delta_{\text{max}}} \), where \( p' \) is the sum of the weight of the passenger paths the algorithm drops before it starts to wait, and \( \frac{T - \delta}{\delta_{\text{max}}} \) is a scaling factor for any given fixed \( \delta \) and \( T \).

Recall that \( \delta_{\text{max}} = w \cdot \delta \), where \( w \) is the total weight of the passenger paths in the instance. So, no matter where the algorithm decides to wait, the overall costs for delaying all passenger paths by \( \delta \) time units are always \( \delta_{\text{max}} \). Further, because all passenger paths travel to the terminal node, the only information that can be used to take a decision is the total weight of the passenger paths that are dropped before starting to wait, possibly compared with the total weight of the passenger paths. In the best case, any decision might additionally be weighted with a function of \( \delta \) and \( T \), which do not influence the essence since we assume them to be known when the instance is first disclosed. The proportion of the weight of source delayed passenger paths with respect to the total weight of the passenger paths can be expressed as the amount of additional \( T - \delta \)-delay that the dropped source delayed passenger paths cause with respect to \( \delta_{\text{max}} \).

Let \( A(s) \) be the class of deterministic online algorithms which wait after having accumulated \( s \cdot \delta_{\text{max}} \) units of \( (T - \delta) \)-delay, that is, when \( (T - \delta)p'_{\delta} \geq s \cdot \delta_{\text{max}} \), where \( p'_{\delta} \) is the total weight of the source delayed passenger paths dropped by \( A(s) \).

**Theorem 4.13.** For the online delay management problem for multiple trains on a terminal railway corridor, the best deterministic online
algorithm of the family $A(s)$ is $s = 1$, which achieves a competitive ratio of 2. Furthermore, 2 is the lower bound on the competitive ratio of any deterministic online algorithm for this setting.

Proof. Consider the deterministic algorithm $A(s_0)$ for any fixed $s_0$, and let $p_\delta$ be the total weight of the source delayed passenger paths, known a priori to the adversary and only a posteriori to the online algorithm. For $s_0 > 1$ the adversary chooses an instance where $(T - \delta) \cdot p_\delta > \delta_{\text{max}}$ (note that this is possible, by choosing $p_\delta = \frac{\delta}{T - \delta} \cdot w + 1$). In this case, the competitive ratio is $c_r = \frac{\delta_{\text{max}} + s_0 \cdot \delta_{\text{max}}}{\delta_{\text{max}}} = 1 + s_0$, since $A(s_0)$ waits after having caused an additional delay of $s_0 \cdot \delta_{\text{max}}$ and the optimal offline algorithm immediately waits. Similarly, if $s_0 \leq 1$, the adversary chooses an instance such that $(T - \delta) \cdot p_\delta = s_0 \delta_{\text{max}} \leq \delta_{\text{max}}$, that is, where it is best not to wait. The competitive ratio is $c_r = \frac{\delta_{\text{max}} + s_0 \cdot \delta_{\text{max}}}{s_0 \cdot \delta_{\text{max}} + s_0 \delta_{\text{max}} \cdot \frac{\delta}{T - \delta}} = \frac{1 + s_0}{s_0 (1 + \frac{\delta}{T - \delta})}$. For $\frac{\delta}{T - \delta} \to 0$ we have $c_r \to 1 + \frac{1}{s_0}$. Considering both cases, the best competitive ratio is achieved for $s_0 = 1$. \qed

Note that even by fixing $T$ and $\delta$ a priori with $T \gg \delta$, it is always possible to enforce the behavior above by setting $p$ and $p_\delta$ accordingly; naturally, if $\delta \approx T$ the analysis for $s_0 \leq 1$ gets worse.

Corollary 4.14. The lower bound on the competitive ratio of the online delay management for a single train on a railway corridor is 2.

This claim immediately follows, since the considered multiple train setting can be interpreted as a single train setting, and the structure of the optimal solution of the first is a solution to the latter setting.

Finally, we observe that the considerations above apply as well to the case where we allow all trains services to have intermediate stops, maintaining the restriction that all passenger paths travel to the terminal node. Indeed, as in Observation 4.5, an optimal solution which does not drop any punctual passenger paths exists here as well. Thus, the obvious extension of the online algorithm to this setting leads a 2-competitive algorithm. Note that the lower bound immediately applies, since the canonical setting above is a special case hereof.
4.5 Extensions and Limitations

This section first presents two extensions to the model discussed so far: randomization of the online algorithm, and an alternative meaningful object function. Next, we reflect on the limitations of our ski rental approaches for dealing with less restricted problem settings.

4.5.1 Randomization using $A(s)$

This section investigates what happens when the online algorithm can use randomization when deciding at which outbound leg to start waiting. We consider the online delay management problem with multiple trains on a terminal railway corridor. We follow the analysis in [32] for the randomized ski rental problem and show that the lower bound on the competitive ratio against an oblivious adversary is $\frac{c}{e-1}$.

Several different types of adversaries have been introduced for randomized online algorithm (see, e.g. [7]). In this section, we consider an oblivious adversary. An oblivious adversary needs to generate the complete request sequence before serving them to the online algorithm. He is told the workings of the online algorithm, but must generate the complete sequence (and stick to it) before the online algorithm preforms its first move. Naturally, the generated request sequence is still disclosed in an online fashion. The cost paid by the adversary in the competitive analysis is that of serving the generated request sequence to optimality.

The notation and the way of operation in the analysis that follows is taken from [32] and adapted to our specific problem to draw the analogy to the ski rental problem using the notation of Section 4.13.

The input disclosed online to the online algorithm by the adversary consists of $p_\delta$, the weight of the passenger paths which are source delayed. From this, we define $u = \frac{(T-\delta)\cdot p_\delta}{\delta_{\max}}$, that is, we norm cost of dropping the passenger paths in addition to their source delay by $\delta_{\max}$. Hence, the input of the adversary is parameterized with respect to $u$.

The cost $C$ of $A(s)$ (as specified before), with a fixed $s$, and of the optimal offline algorithm OPT on the adversarial input $u$ are given by, respectively,
Chapter 4. Renting skis and managing delays online

\[ \begin{align*}
C(A(s), u) &= \begin{cases} 
u \cdot \delta_{\text{max}} \left(1 + \frac{\delta}{T - \delta}\right) & \text{if } u \leq s \\ 
\delta_{\text{max}} \cdot (1 + u) & \text{if } u > s 
\end{cases} \\
C(\text{OPT}, u) &= \min \left\{ \delta_{\text{max}}, \nu \cdot \delta_{\text{max}} \left(1 + \frac{\delta}{T - \delta}\right) \right\} 
\end{align*} \]

As for the ski rental problem, it is reasonable for any online algorithm to wait no later than when \( s = 1 \). Hence, the interesting intervals for both \( u \) and \( s \) lie between 0 and 1. Thus, the competitive ratio \( c_r(u) \) is

\[ c_r(u) = \frac{C(A(s), u)}{C(\text{OPT}, u)} = \begin{cases} 
\frac{u \cdot \delta_{\text{max}} \left(1 + \frac{\delta}{T - \delta}\right)}{u \cdot \delta_{\text{max}} \left(1 + \frac{\delta}{T - \delta}\right)} = 1 & \text{if } u \leq s \\
\frac{\delta_{\text{max}} \cdot (1 + u)}{\delta_{\text{max}} \left(1 + \frac{\delta}{T - \delta}\right)} \rightarrow 1 + \frac{1 + s}{u} & \text{if } s < u \leq 1.
\end{cases} \]

For \( T \gg \delta \), this leads to the exact same cases as for the ski rental problem. Thus, the analysis of [32] gives the best competitive ratio for a randomized online algorithm against an oblivious adversary of \( \frac{e}{e - 1} \) with a probability distribution \( p(s) \) on \( A(s) \) of \( p(s) = \frac{e^s}{e - 1} \).

4.5.2 Lower bounds for the additional passenger delay objective

So far, the objective function for our delay management problems was to minimize the total passenger delay. This objective also counts the source delay \( \delta \) of the source delayed passenger paths, which cannot be avoided by any delay policy.

Here, we consider the additional passenger delay objective function in the context of the online delay management problem with a single train on a railway corridor. The additional passenger delay objective function only accounts for the delay which can be optimized on the network, that is, without counting the source delays. Using the notation from Section 4.2.1, the value of the additional passenger delay objective \( \Delta_{\text{ADD}}(j) \) occurring if the algorithm starts waiting at the outbound leg from node \( j \) is given by

\[ \Delta_{\text{ADD}}(j) = (T - \delta) \sum_{i < j} D^i + \delta O_{\geq j}. \]
Figure 4.5.1: The railway network used for showing the non-competitiveness of the additional passenger delay objective. Passenger path $P^1$ has a source delay $\delta$, whereas the source delay of $P^2$ is still unknown.

Theorem 4.15. For online delay management with a single train on a railway corridor, no online algorithm can have a competitive ratio better than $\frac{T}{\delta}$ when minimizing the additional passenger delay.

Proof. We analyze the railway network shown in Figure 4.5.1. The train service has source in $A$ and destination in $C$, and has an intermediate stop in $B$. Two passenger paths wish to board this train service. $P_1$ has origin in $A$ and weight $w(P_1) = 1$, $P_2$ has origin in $B$ and weight $w(P_2) = \frac{T(T - \delta)}{\delta^2}$.

Initially, the adversary discloses $P_1$ as source delayed. He can still choose if or not he source delays $P_2$. If the online algorithm decides to wait at $A$, the adversary leaves $P_2$ source punctual. Thus, the optimal offline policy is to make all legs depart on time. The delay accumulated by the online algorithm is $\frac{T(T - \delta)}{\delta^2} \delta = \frac{T(T - \delta)}{\delta}$, the optimal strategy accumulates only $(T - \delta)$ delay. Hence, the online algorithm is $\frac{T}{\delta}$-competitive.

If the online algorithm decides to leave $A$ on time, the adversary discloses $P_2$ as source delayed. The optimal offline policy in this case is to wait in $A$, and this policy produces a zero valued objective. The online algorithm, however, produces an additional passenger delay of at least $T$, hence the competitive ratio in this case is infinite.

It follows that no online algorithm for this setting can have a competitive ratio better than $\frac{T}{\delta}$ when optimizing the additional passenger delay.

Note that a competitive ratio of $\frac{T}{\delta}$ is useless, as it can on the one hand be made arbitrarily big by choosing $T \gg \delta$, and implies that
the algorithm inflicts an additional waiting time of dropping a set of passengers which an optimal strategy would only delay by $\delta$.

The setting for proving the $\frac{T}{\delta}$-competitiveness might seem peculiar, as in one case the leg travels empty between nodes A and B. This problem can be resolved by introducing a source punctual passenger path between A and C with unit weight. The same setting can then be used to prove a lower bound on the competitive ratio of $\frac{T}{\delta} - 1$. This is asymptotically the same as what we have proved above, and in practice it does not change significantly if $\frac{T}{\delta}$ is large.

### 4.5.3 Limits of ski-rental-like approaches

In the following, we show how ski-rental-like approaches fail for instances on railway corridor and passenger paths which are unrestricted in terms of destination node. Thus, we consider instances of the online delay management problem for multiple trains on a railway corridor. Here, we again focus on our classical objective function, the total passenger delay. To that end, we describe several natural ski rental-like strategies and give an example for each which leads to an unsatisfactory competitive ratio. For delay management, a competitive ratio is unsatisfactory if it either depends on the input size, or exceeds the ratio $\frac{T}{\delta}$. The latter applies since it means that the algorithm inflicted the delay of dropping all delayed passenger paths when it would have been possible to only to delay them.

For ease of reading, we define the dropping costs of a set of passenger paths as the sum of their weights multiplied by $(T - \delta)$, and refer to the cost of delaying a set of passenger paths by $\delta$ as the waiting costs, defined as the sum of the passenger path’s weights multiplied by $\delta$. We point out that the dropping costs are defined arbitrarily as $(T - \delta)$, and all constructions can be adapted to dropping costs in the interval $[T - \delta, T]$ without loosing the stated competitive ratio. We focus on $T - \delta$ for ease of exposition.

**A ski rental like approach for each leg**

A first option is to apply the ski rental scheme to each individual leg. Specifically, an online algorithm could compute, at each node of the corridor, the dropping costs of the delayed passenger paths which have not already been dropped and wish to board that leg, and
Figure 4.5.2: The network where a ski rental like approach for each leg fails. Arrows represent the legs. The dashed line is the passenger path of weight one, which is disclosed as source delayed by the adversary; solid lines are passenger paths of weight \( \beta = \frac{T-\delta}{\delta} - \epsilon \) which the adversary sequentially discloses as source punctual with the given online algorithm.

compare this cost with the waiting costs of all passenger paths that would be delayed by its decision to wait. Larger dropping costs cause the algorithm to wait.

This straight forward approach fails badly already for this simple network structure, and even for the case where all but one passenger path use a single leg, as we show next. The precise configuration of an instance achieving a bad competitive ratio is depicted in Figure 4.5.2. A passenger path of unit weight spans through the whole network. Additionally, each leg is traveled by a passenger path using only that leg and with weight \( \beta = \frac{T-\delta}{\delta} - \epsilon \).

The adversary presents this instance, and lets the unit weight passenger path be source delayed, whereas he lets the passenger path of weight \( \beta \) be source punctual. Clearly, the online algorithm makes the first leg wait, as \( T - \delta \geq (\frac{T-\delta}{\delta} - \epsilon) \delta \). At each further step, the adversary discloses the remaining passenger paths as source punctual. Hence, the online algorithm takes the same decision at each leg, since in this narrow minded approach each leg looks the same as the first leg. Therefore, the online algorithm incurs in a cost of \( n \frac{T-\delta}{\delta} + n \epsilon \delta + \delta = nT - \delta (n - n \epsilon + 1) \).

The optimal offline algorithm, on the contrary, drops the source delayed passenger path and has a cost of \( T \). By setting \( \delta = 1, n \epsilon = 1 \) and \( T \to \infty \) we get the competitive ratio of

\[
C_{\text{One}} = \frac{nT - n \delta}{T} = n - n \frac{\delta}{T} \to n.
\]
Chapter 4. Renting skis and managing delays online

Figure 4.5.3: The network where an averaged ski rental like approach fails. The network representation is as in Figure 4.5.2, with
\[ w' = \frac{T-\delta}{2\delta} - \epsilon. \]

An averaged ski rental like approach for each leg

One of the reasons the previous approach fails is that it maintains an extremely local view. As a remedy, one can divide the cost of dropping the passenger paths by the number of legs used by the passenger path, in an effort to average the impact of dropping a passenger path over each of the used legs. To be precise, at each node the algorithm compares the dropping costs of the delayed passenger paths wishing to board the outbound leg, each dropping cost divided by the number of legs used by the respective passenger path, with the waiting costs the decision would inflict to the other involved passenger paths using that leg. Again, if the latter costs exceed the dropping costs, the algorithm makes the leg wait.

The averaging renders the construction in Figure 4.5.2 useless. However, a similar counterexample is sufficient to prove a bad competitive ratio. The passenger paths defined on the railway network which leads to a bad competitive ratio are shown in Figure 4.5.3.

The adversary discloses the unit weight passenger path as source delayed when the online algorithm considers the first leg, and lets the other passenger path boarding the leg be source punctual. The online algorithm compares the averaged dropping costs (of the unit weight passenger path) with the waiting costs (of the passenger path of weight \( w' \)): \[ \frac{T-\delta}{2\delta} > \delta \left( \frac{T-\delta}{2\delta} - \epsilon \right), \]
and hence it makes the first leg wait. At each of the subsequent nodes the adversary discloses the involved passenger paths as source punctual. Hence, since the weights of the involved passenger paths are the same, the online algorithm takes the same decision (to wait) also for the second leg of the corridor. At each subsequent node, under the reasonable assumption that \( T - \delta > \delta \), it is obviously costlier to drop a passenger path of weight \( w' \) rather than delaying a passenger path of weight \( w' \):
\[ \left( \frac{T-\delta}{2\delta} - \epsilon \right) \frac{T-\delta}{2} > \left( \frac{T-\delta}{2\delta} - \epsilon \right) \delta. \]
Hence, the online algorithm makes
4.5. Extensions and Limitations

Figure 4.5.4: The bad example for a global ski rental like approach. The network representation is as in Figure 4.5.2.

all legs wait, causing overall costs of \( n \frac{T-1}{2} - n\epsilon \delta + \delta \) for the online algorithm. The optimal offline algorithm with these source delays, on the other hand, drops the unit weight passenger path and incurs in a cost of \( T \). For \( n\epsilon = 1 \) and \( T \gg 1 \), the competitive ratio is

\[
C_{\text{Averaged}} = \frac{n \frac{T-1}{2} - n\epsilon \delta + \delta}{T} \rightarrow \frac{n}{2}.
\]

Note that this example leads to the same negative result if the waiting costs are also divided by the number of legs used by each passenger path.

A global ski rental like approach for each leg

Finally, one could apply the ski rental like approach globally, as done for the online delay management problem for multiple trains on a terminal railway corridor, and compare the overall dropping costs with the overall waiting costs. Note that the negative example of Figure 4.5.2 also settles the case where we compare the overall dropping costs with each leg’s waiting costs.

To be precise, the online algorithm compares the dropping costs of all source delayed passenger paths with origin node smaller than the source node of the leg it has to decide of, and compares them with the cost of delaying all passenger paths, that is, the waiting costs of the affected passenger paths if all subsequent legs wait.

The simple network in Figure 4.5.4 is sufficient to show that such an approach does not lead to a successful competitive ratio. Each leg is used by exactly one passenger path. The passenger path \( P_{w''} \) willing to board the first leg of the corridor has weight \( w'' \), all other passenger paths have weight \( w' \). Now, the exact size of these weights is irrelevant. It is sufficient that if \( P_{w''} \) is source delayed, its dropping costs do not exceed the sum of the waiting costs of the remaining \( n - 1 \) passenger paths of weight \( w' \).
Figure 4.5.5: The adaptation of the network of Figure 4.5.4. Dotted passenger paths are passenger paths with very small weight with respect to the rest of the passenger paths, such that any decision on them does not affect the optimum significantly nor the behavior of the online algorithm.

Now, the adversary discloses $P_{w''}$ as being source delayed. As $T_{w''} < (n - 1)\delta w'$, the online algorithm does not make the first leg wait for $P_{w''}$. In a second step, the adversary may disclose the information about all the remaining passenger paths and set them as punctual. Now, it is irrelevant what the subsequent choices of the online algorithm are, since the online algorithm has already incurred in $T_{w''}$ delay, whereas the optimum offline policy only incurs in a cost of $\delta w'$, since it makes the first leg wait and the remaining legs depart on time. Thus, we get a competitive ratio of

$$C_{\text{Global}} \geq \frac{T}{\delta}.$$ 

One might argue that all the above examples are extremely artificial, since there is little interaction between the different passenger paths. In particular, the example in Figure 4.5.4 leaves each leg basically independent. This problem can easily be solved by scaling all the weights of the passenger paths by a big constant, and introducing unit weight passenger paths which interconnect the different legs of the network, see Figure 4.5.5 for an example of the adaptation for the network of Figure 4.5.4. Furthermore, the given examples just point out the weaknesses of each online policy in a glaring way. Naturally, since these algorithms fail already on the simple network topology of a railway corridor, and for train services without intermediate stops, these algorithms are doomed to failure if we apply to more complex settings, as the setting considered above is a special case thereof.
Chapter 5

Robust Online Scheduling

Earl: I’m putting you on the List.
Randy: what?

[···]
Earl: Randy. You’re going on the List, Randy.
Randy: I don’t wanna.
Earl: It’s not up to you. Its up to Karma.

My name is Earl, Season 1, Episode 11.

In the previous chapters, we have taken a reactive approach to handle uncertainties. Indeed, the main question we have addressed is what actions should be taken to guarantee that the quality of the solution at hand (a timetable) does not degrade too much. In this chapter, we take a different view of perturbations. Here, we are interested in how much perturbations degrade the quality of a solution that cannot be adapted when the perturbations arise. The motivation for analyzing this setting comes from real-world problems: many of these problems are optimized with respect to data which is not always accurate. For instance, consider the traveling times of trains between stations: these times are only estimates on the actual traveling times; due to severe weather conditions or engine breakdowns, the actual traveling times might turn out to be much longer than estimated. However, the railway operator might not be able to adapt the computed solution of, say, the track assignment, since some trains might already be using the tracks and cannot turn around. Hence, the quality of the solution
degrades without the operator being able to do anything. Thus, we have a two-phase problem: in the first phase, an algorithm computes a solution for the given instance of a problem, which we call the original instance. In the second phase, the perturbations are revealed, and affect the quality of the solution computed on the original instance. The goal is to bound the impact on the quality of the solution computed in the first phase when evaluated on the perturbed data given in the second phase.

We take a step in this direction by addressing the tolerance to perturbations of Graham’s algorithm for the simple online parallel machine scheduling problem. We focus on an online setting as a setting where decisions cannot be revoked, and consider parallel machine scheduling with Graham’s algorithm as a simple example of an algorithm which is in general quite well understood. The input to the parallel machine scheduling problem is a sequence of processing times for $n$ jobs, each of which needs to be processed later on one of $m$ identical machines. The objective is to minimize the completion time of the last terminating job (the makespan) $[50]$. A feasible solution is an assignment of jobs to machines; an optimal solution minimizes the makespan. The underlying assumption is as usual for all kinds of (offline or online) optimization problems: The input values accurately reflect the reality for which the planning takes place. In online parallel machine scheduling, the planning decision for each job, i.e., to which machine this job shall be assigned, must be made when the job is presented, before the next processing time (if any) is shown, and cannot be revoked later. Note that there is a clear distinction between the presentation of the processing times of the jobs for the purpose of planning, and processing the jobs on the planned machines. Therefore, the planned processing time of a job can sometimes be dramatically different from its real processing time, when processing takes place.

Here, we aim at understanding to what extent a limited number of perturbations of the processing times affect the performance of online parallel machine scheduling. We assume that a probability distribution of perturbations is either not available, or does not tell a lot (as for engine breakdowns, which with very small probability may occur everywhere), and focus on a worst-case analysis. We study the behavior under perturbations of the well-known algorithm by Graham $[26]$ for online scheduling, the List scheduling algorithm. Graham’s algo-
algorithm assigns the next presented job to the machine that will terminate earliest for the job sequence seen so far. Although online algorithms which improve the competitive ratio exist, Graham’s algorithm remains, in its simplicity and with its competitive ratio of $2 - \frac{1}{m}$, a prime example of a good online algorithm.

The perturbations of the processing times are disclosed after the entire solution has been determined. The impact of perturbations is measured by the worst-case ratio of two makespans. The first is the makespan that we get by taking Graham’s assignment of jobs to machines for the original instance, and by replacing, within this assignment, all original processing times with the perturbed ones. The second is the makespan of an optimal offline solution of the perturbed instance. This ratio tends to be larger than Graham’s competitive ratio; the larger it is, the more Graham’s algorithm suffers from perturbations. Our performance measure is not applicable to all online problems, since a solution computed for the unperturbed instance might not be feasible for the perturbed instance. This happens if, for example, the problem has some resource constraint which can be violated by the perturbation. For parallel machine scheduling, however, any assignment of jobs to machines is feasible.

A related but different question has been addressed in sensitivity analysis. Sensitivity analysis asks for the amount of perturbation that can be tolerated before the structure of an optimal solution changes. Sensitivity analysis has been studied predominantly for linear programming [14], but also for many combinatorial problems such as network flows [2] and scheduling [28]. We point out that an instance which is unstable with respect to sensitivity analysis does not necessarily need to be a bad instance in our worst-case setting. Indeed, although the structure of the optimal solution might change with a very small perturbation, the quality of the solution of the unperturbed optimum can remain very near to the one of the optimum of the perturbed instance with these perturbations. Throughout this chapter, we allow a small number of processing times to be perturbed a lot, but we will not change the structure of a solution and merely observe its change in quality, because the perturbation happens after the irreversible assignment decision.

This setting can also be interpreted as introducing a more powerful adversary with respect to the well-known online setting. Not only must each decision be taken immediately each time the adver-
sary presents a new slice of the instance, but the adversary may even lie on some of the information he gives. With this view, the adversary can choose a sequence which is worst-case both from an online point of view and from the point of view of the perturbations, and lie at crucial stages when disclosing the input. Naturally, the extent of the adversary’s lies greatly affect the performance of online algorithms which are not tuned for lies. Indeed, we show that Graham’s scheduling algorithm performs better for bounded perturbations than for unbounded ones.

Related work

Online scheduling on identical parallel machines has been studied extensively, see [58] for a survey. The first deterministic online algorithm by Graham [26], the List scheduling algorithm, achieves a competitive ratio of $2 - \frac{1}{m}$. This competitive ratio was improved several times over the years, and the to-date best deterministic online algorithm by [19] achieves a competitive ratio of 1.9201. Randomized algorithms have also been widely addressed, as for instance in [3].

Robustness has been defined in a variety of ways in optimization theory. Exact input values have been replaced by probability distributions [53] in a probabilistic approach, or by intervals of possible values in a worst case scenario [43, 34], or by uncertainty sets [8]. We are most interested in a notion of robustness that takes a worst-case view, but limits the number of perturbations that can actually happen. This view is motivated by everyday experience: A severe disturbance such as a locomotive breakdown can happen to any engine, but we do not need to fear many such events simultaneously.

The effect of perturbations in offline scheduling algorithms has been studied under many different points of view, see [28, 38] for an overview. A sensitivity analysis for different parameters is given in [28]. The analysis also addresses how perturbations affect the objective, and how to reconstruct an optimal solution given some perturbations.

The variant where the jobs are scheduled on a single machine according to some priorities and each job has a release time has also been considered [40]. There, the release times are perturbed, and the goal is to efficiently reconstruct a feasible schedule.

The scheduling variant where binary precedence relations exist
between jobs has been addressed as follows in the context of robust-
ness. Consider two jobs scheduled on two different machines that
have a direct precedence relation. Then, there is a communication
delay before the second job can start being processed, since the re-
sult of the first job must be transferred between the two machines.
The uncertainty lies in the actual size of the communication delay,
which may be given as an interval of possible values. For this set-
ting, [52] gives an analysis on which estimates an algorithm should
use in order to produce a stable solution. For two machines and re-
stricted precedences, [44] provides an algorithm with an absolute per-
formance guarantee with respect to the optimum.

The effect of perturbed processing times has also been addressed
in different ways. Parallel machine scheduling where all jobs’ pro-
cessing times are accurate up to a factor \((1 \pm \varepsilon')\) of a declared value
was analyzed in [49]. The quality of any algorithm deteriorates by a
factor \((1 + \varepsilon)\) for the makespan, and by \(\sqrt{1 + \varepsilon}\) for the sum of com-
pletion times. We provide a better bound for the makespan, since
our analysis exploits the structure of Graham’s schedule. The per-
formance of online scheduling have also been addressed with respect
to processing times drawn from a distribution. With this view, the
scheduling algorithm needs to schedule the jobs with the knowledge
of the distributions only. This setting has been addressed in terms of
average case analysis for the completion times [53] and of minimiza-
tion of an objective in expectation, as in [42].

The tolerance to perturbations of Graham’s offline List scheduling
algorithm has been addressed both in terms of decrease in quality of
the objective [26, 27] and in the number of different (offline) sched-
ules that arise from perturbing the processing time of one job and for
different input sequences [37]. The List scheduling algorithm was
presented for settings more complex than parallel machine schedul-
ing, which also included precedence constraints. Graham [26, 27]
analyzed the effects of relaxating a number of instance-defining pa-
rameters, as for instance the number of machines or the precedence
constraints. For each parameter, he analyzed the worst-case ratio of
the makespans of the relaxed instance with respect to the non-relaxed
instance, for both instances scheduled with List. He showed a worst-
case ratio of \(2 - \frac{1}{n}\) for decreasing the processing times, and similar
results were derived for the relaxation of other parameters.
Chapter 5. Robust Online Scheduling

Summary of results

The results of this chapter are joint work with Peter Widmayer.

We derive lower bounds on the competitive ratio of Graham’s algorithm for the perturbed schedule for following scenarios: For integer $r$, increase the processing times of $r \leq n$ jobs arbitrarily; arbitrarily decrease the processing times of any number of jobs scheduled on $r \leq m$ machines in an optimal offline schedule; for $x > 1, x \in \mathbb{Q}_0^+$, divide the processing times of $r$ jobs by a factor $x$; and either divide or multiply (but not both) the processing times of an arbitrary number of jobs by a factor $x > 1$. We also give infinite families of examples where the bounds are tight or come close. Our results imply specific bounds for specific cases: If the processing times of three jobs increase arbitrarily, Graham’s algorithm is $(5 - \frac{6}{m})$-competitive; if the processing time of one job decreases arbitrarily it is $2$-competitive; and if jobs may triplicate their processing times, it is $(4 - \frac{1}{m})$-competitive.

We also provide simple lower bounds on the competitive ratio for several settings of perturbations. We show that no online algorithm can have a competitive ratio smaller than $2$ if the perturbation may decrease the processing time of one job arbitrarily. For the setting where the perturbation may decrease the processing time of one job to a factor at least $\frac{1}{x}, x > 1, x \in \mathbb{Q}_0^+$, of its original processing time, we show a lower bound of $2 - \frac{2}{x+1}$; for the case of perturbations decreasing the processing time of at most two jobs to a factor at least $\frac{1}{2}$ of their original processing time, we show a lower bound of $2$. For the case of the perturbations increasing the processing time of one job to a factor at most $2$ of its original processing time, we show a lower bound of $\frac{3}{2}$, and for the case of perturbing two jobs to a factor at least $2$ of their original processing time we show a lower bound of $2$.

Finally, we propose a Graham-like algorithm designed to be robust with respect to the possibility of an $x$-fold increase of the processing time of one job, for $x > 2, x \in \mathbb{Q}_0^+$. We show that the algorithm is $x$-competitive if no perturbation arises, and $1 + x - \frac{1}{x}$ if a perturbation arises, thus performing much worse than Graham’s algorithm. We also give examples which come close to these bounds.
5.1 Problem setting and notation

The problem is specified as a 3-tuple \((J, P, \tilde{P})\), where \(J\) is the online sequence of \(n\) jobs to be scheduled on \(m\) machines, \(P\) and \(\tilde{P}\) specify the original processing time \(p_j \in \mathbb{Q}_0^+\) and the perturbed processing time \(\tilde{p}_j \in \mathbb{Q}_0^+\) of each job \(j \in J\), respectively. A schedule for the sequence of jobs is an assignment of the jobs to the machines. Graham’s algorithm (which we briefly recall at the end of this section) schedules the original instance of the online parallel machine scheduling problem, that is, the job sequence \(J\) with processing times \(P\), and produces a schedule \(\text{List}(J, P)\).

Each problem is characterized by the perturbation against which we analyze robustness. The effect of the perturbation is reflected in the processing times \(\tilde{P}\), which may either increase or decrease; the perturbation may be of arbitrary size, or bounded, for each job, by a factor \(x\) of the job’s original processing time. The latter setting is motivated by project scheduling, where the extent of the misjudgment of a task’s processing time is often linked to the task’s difficulty. We refer to the jobs \(J\) with processing times \(\tilde{P}\) as the perturbed instance of the online parallel machine scheduling problem.

We denote by \(\text{OPT}(J, \tilde{P})\) the optimal offline schedule of the sequence of jobs \(J\) with processing times \(\tilde{P}\). In the offline setting, the order in \(J\) is irrelevant. The makespan of a schedule is the time when the last terminating job finishes being processed. We denote by \(\mathcal{L}(S, P)\) the makespan of the schedule \(S\) with processing times \(P\). In this setting, we measure the robustness of Graham’s online algorithm by considering the makespan of the schedule \(\text{List}(J, P)\) obtained with the original instance, but evaluated on the perturbed processing times \(\tilde{P}\). Hence, we evaluate \(\mathcal{L}(\text{List}(J, P), \tilde{P})\). We compare this makespan with the makespan \(\mathcal{L}(\text{OPT}(J, \tilde{P}), \tilde{P})\) of an optimal offline schedule \(\text{OPT}(J, \tilde{P})\). Note that comparing \(\mathcal{L}(\text{List}(J, P), \tilde{P})\) with \(\mathcal{L}(\text{List}(J, P), P)\) does not provide any useful information, since the total processing time is different for \(P\) and \(\tilde{P}\): If the processing time of a job increases arbitrarily, any algorithm needs to process this job.

In this model, each machine processes its assigned jobs without pausing in between. The sum of the processing times of a machine is called load. When machine \(i \in M\) is finished, it remains idle up to the makespan. We refer to the idle time as \(s_i, i \in M\). We call the set of
machines which process some perturbed jobs the affected machines, and denote them by $M\neq$. Similarly, we call the set of machines which do not process any perturbed jobs unaffected machines, and denote them by $M=.$

For perturbations increasing the jobs’ processing times, we denote the perturbed processing times as $p_j^+, j \in J$, and the set of all increased processing times as $P^+$. Similarly, for decreases we use $p_j^-, j \in J$ and $P^-$. Changing the processing times of the jobs in a schedule also influences the idle times. Therefore, we refer to the idle time resulting after a perturbation as $s_i^+$ for increased processing times and as $s_i^-$ for decreases. In the analysis, we look at the perturbations of the jobs sequentially, in any order. In this way, we can specify the impact of the perturbation of each job on the idle time of each machine. When a job changes its processing time, the subsequent jobs shift accordingly in the schedule. This shift may shorten or lengthen the idle time of various machines. We denote the increase or decrease in idle time on machine $i \in M$ caused by perturbing job $j$ by $\delta_j^i \in \mathbb{Q}$, which may be positive or negative.

We recall Graham’s List scheduling algorithm for the online parallel machine scheduling problem (i.e., without perturbations). When considering the next job of the online sequence, Graham’s algorithm schedules it on a machine having least load given the previous assignments of jobs to machines. The classical competitive analysis of Graham’s algorithm is as follows. For a makespan $L_{\text{List}}$ obtained with Graham’s algorithm, the processing times of the jobs and the idle times of the $m$ machines satisfy $mL_{\text{List}} = \sum_{j \in J} p_j + \sum_{i=1}^m s_i$. Let $L_{\text{OPT}}$ be the optimal makespan (for unperturbed processing times). As each job must be scheduled non-preemptively by any algorithm, $p_j \leq L_{\text{OPT}}, \forall j \in J$. Furthermore, $L_{\text{OPT}} \geq \frac{\sum_{j \in J} p_j}{m}$, since no schedule can do better than distribute the total processing time evenly across all machines. Finally, consider an arbitrary job $j$ finishing at the makespan of Graham’s schedule. Then, $s_i \leq p_j, \forall i \in M$, since otherwise Graham’s algorithm would have scheduled job $j$ on the machine not satisfying the inequality. Furthermore, the machine attaining the makespan has zero idle time. Thus: $mL_{\text{List}} = \sum_{j \in J} p_j + \sum_{i=1}^m s_i \leq mL_{\text{OPT}} + (m-1)p_j \leq (2m-1)L_{\text{OPT}}$. This proof shows a competitive ratio of $2 - \frac{1}{m}$ for Graham’s algorithm.
5.2. Arbitrary perturbations

Finally, consider the (perturbed) instances where the perturbation increases the processing times. For these cases, $L(OPT(J, P), P) \leq L(OPT(J, P^\uparrow), P^\uparrow)$ holds, since the total processing time increases and the maximum processing time of the jobs can also only become larger.

To improve the readability of our proofs, we use the following compact notation for the makespans. Instead of $L(OPT(J, P), P)$ we write $L_{OPT}$, and instead of $L(OPT(J, P^\downarrow), P^\downarrow)$ we write $L_{OPT}^\downarrow$. Similarly, we write $L_{List}^\downarrow$ instead of $L(List(J, P), P^\downarrow)$ for Graham’s algorithm. The notation for increases in processing times is obtained accordingly.

5.2 Arbitrary perturbations

In the following, we analyze robustness for arbitrary size perturbations of the processing times of jobs. First, we analyze the case of arbitrary decreases in processing times and then of arbitrary increases.

5.2.1 Arbitrary decreases in processing time

As a first step, we bound the quality of the solution of any best-possible online algorithm if the processing times may decrease arbitrarily.

**Theorem 5.1.** No algorithm for online scheduling on $m \geq 2$ identical parallel machines on instances where the processing time of one job may decrease arbitrarily can have a competitive ratio smaller than 2.

**Proof.** Assume such an algorithm exists, and consider the following sequence of jobs, all with processing time 1. First, the adversary sequentially presents $m$ jobs. To be strictly better than 2–competitive, any algorithm must schedule each job on a different machine. Then, the adversary presents a final job, which may be scheduled on any machine. Now, the perturbation affects one job which is scheduled alone on a machine, and decreases its processing time to 0. Thus, the computed schedule has an empty machine, and a makespan of 2. The optimum offline perturbed schedule assigns each of the now $m$ jobs on a different machine, and has a makespan of 1. □
Note that the construction can be extended to any number of jobs by introducing an appropriate number of jobs of processing time $\varepsilon \to 0$ which do not influence the construction and allow for a bound arbitrarily close to 2.

We start the analysis of the robustness of Graham’s algorithm by bounding the effect of decreasing the processing time of one job, and extend it to the case where the perturbation may decrease the processing time of many jobs arbitrarily, but these are scheduled on $r$ machines in Graham’s schedule.

**Decreasing the processing time of one job arbitrarily**

If one job decreases arbitrarily in processing time, we get the following theorem, which implies that the loss of one machine averages out on the remaining machines:

**Theorem 5.2.** Graham’s algorithm for online scheduling on $m \geq 2$ identical parallel machines is 2-competitive on instances where the processing time of one job may decrease arbitrarily. Thus, for these instances Graham’s algorithm is optimal.

**Proof.** We distinguish two cases. First, we assume the perturbed job is scheduled on a machine $\tilde{i}$ attaining the makespan after the perturbation. This case implies that the affected machine attained the makespan also before the perturbation took place. In this case, the competitive ratio remains unchanged, since the complete analysis of Graham’s algorithm carries through. In particular, the idle times of all machines are still not bigger than the processing time of the last job scheduled on $\tilde{i}$, since the perturbation slides the completion time of some jobs earlier in time, thus globally shortening the makespan as the jobs are always processed seamlessly.

Now, we analyze the case where the perturbed job is not scheduled on a machine attaining the makespan after the perturbation has taken place. Let $\tilde{j}$ be the job which was perturbed. Let $M^= = M \setminus \{\tilde{i}\}$ be the set of unaffected machines. Furthermore, let $J^=$ be the set of jobs which are scheduled on $M^=$, and have thus all maintained their original processing time.

We bound the impact of decreasing one job’s processing time by analyzing the time spent by the unaffected machines $M^=$ up to the makespan. Note that restricted on $M^=$ and $J^=$, Graham’s algorithm
works exactly as usual, that is, each job within $J^-$ is scheduled on the machine within $M^-$ which, with the current schedule, finishes earliest. Now, let $j_\mu \in J^-$ be the job scheduled last on a machine $\mu \in M^-$ that attains the makespan after the perturbation has taken place. Also the optimal offline algorithm must schedule $\bar{j}_\mu$. Hence, $p_{\bar{j}_\mu} = p_{\bar{j}_\mu} \leq L_{\text{OPT}}^\downarrow$, and the idle times after the decrease satisfy $s_i \leq p_{\bar{j}_\mu} \leq L_{\text{OPT}}^\downarrow, i \in M^==$. We have:

\begin{align}
(m - 1) \cdot L_{\text{List}}^\downarrow &= \sum_{j \in J^-} p_j^\downarrow + \sum_{i \in M^-} s_i^\downarrow,
\end{align}

\begin{align}
\leq \sum_{j \in J^-} p_j^\downarrow + (m - 2) \cdot p_{\bar{j}_\mu}^\downarrow
\end{align}

\begin{align}
\leq m \cdot L_{\text{OPT}}^\downarrow + (m - 2) \cdot L_{\text{OPT}}^\downarrow.
\end{align}

Note that in inequality (5.2.2) we bound the processing time of the $m - 1$ machines by the whole processing time after the decrease, and that the $(m - 2)$ factor arises since at least one of the $m - 1$ machines we consider attains the makespan. Therefore, the makespan is bounded by

\[ L(\text{List}(J, P), P^\downarrow) \leq \frac{2m - 2}{m - 1} L_{\text{OPT}}^\downarrow = 2 \cdot L(\text{OPT}(J, P^\downarrow), P^\downarrow). \]

The optimality of Graham’s algorithm follows from Theorem 5.1.

An infinite family of general instances achieving this bound on $m$ machines is as follows: first, the adversary presents one big job of processing time two. Next, the adversary presents $(m - 1) \cdot m$ small jobs with processing time $\frac{1}{m}$. Graham’s algorithm schedules $m$ of these jobs on each of the $m - 1$ machines which do not process the big job. Finally, the adversary presents a job of processing time 1, which is scheduled on an arbitrary machine processing small jobs, since they all finish earliest. Now, the perturbation decreases the processing time of the first job to zero. Thus, Graham’s algorithm produces a schedule with makespan 2. The optimal offline schedule achieves a makespan of 1 by scheduling the last job alone on one machine and the small jobs evenly on the remaining $m - 1$ machines.

Note that the sequence of small jobs of the tight example needs only to have the property that the jobs are scheduled in such a way that the $m - 1$ remaining machines process a load of exactly one before the adversary presents the last job of processing time 1.
Arbitrary decreases affecting \( r \) machines

We extend the analysis above and derive a bound on the competitive ratio and a worst-case instance which matches the bound for the case where the arbitrary perturbations affect \( r \) machines, that is, where \( r \) machines of Graham’s schedule process jobs whose processing time is perturbed. We have the following theorem:

**Theorem 5.3.** Consider the instances of online scheduling on \( m \geq 2 \) identical parallel machines where perturbations may decrease the processing times of some jobs arbitrarily. Restricted to these instances, if Graham’s algorithm schedules the perturbed jobs on \( r < m \) machines, Graham’s algorithm is \( 2 + \frac{r - 1}{m - r} \)-competitive, and this bound is best possible.

**Proof.** Consider Graham’s schedule \( \text{List}(J, P) \). We refer to the jobs scheduled on the unaffected machines \( M^- \) as \( J^- \). Hence, by definition, these jobs do not change their processing time. To estimate the makespan after the perturbation we analyze the schedule of the \( m - r \) unaffected machines \( M^- \). We distinguish two cases: in the first, the makespan \( L_{\text{List}}^\downarrow \) is attained by at least one machine in \( M^- \), while in the second no machine in \( M^- \) attains it.

For the first case, we let \( \bar{j}_\mu \) be a job attaining the makespan after the perturbation on an unaffected machine \( \mu \in M^- \). In this case, the time spent by the machines \( M^- \) up to the makespan is given by:

\[
(m - r) \cdot L_{\text{List}}^\downarrow = \sum_{j \in J^-} p_j^\downarrow + \sum_{i \in M^-} s_i^\downarrow \\
\leq \sum_{j \in J} p_j^\downarrow + (m - r - 1)p_{j_\mu}^\downarrow \\
\leq m L_{\text{OPT}}^\downarrow + (m - r - 1)L_{\text{OPT}}^\downarrow,
\]

since \( J^- \subseteq J \) and at most \( (m - r - 1) \) machines have some idle time, which due to the workings of Graham’s algorithm is smaller than \( p_{j_\mu}^\downarrow \). Thus, the bound follows:

\[
L(\text{List}(J, P), P^\downarrow) \leq \left( 2 + \frac{r - 1}{m - r} \right) \cdot L(\text{OPT}(J, P^\downarrow), P^\downarrow)
\]

For the second case, let \( \bar{j}_\mu \) be a job attaining the makespan after the perturbation on an affected machine \( \mu \in M^\neq \). Because of Graham’s
algorithm, when \( \bar{j}_\mu \) was originally scheduled, \( \mu \) was a machine with least load, say with load \( \ell \). Since the processing times on unaffected machines \( M^- \) remain unchanged, the latter machines have load at least \( \ell \) after the decreases. Thus, their idle time is \( s^i \leq p^i_{j\mu} \), \( i \in M^- \), since \( \bar{j}_\mu \) attains the makespan. Note that this bound holds both if \( \bar{j}_\mu \) is perturbed or it remains unchanged. Thus, the time spent by the machines in \( M^- \) up to \( L^- \) can be represented as follows:

\[
(m - r) \cdot L^{1}_{\text{List}} = \sum_{j \in J^-} p^i_j + \sum_{i \in M^-} s^i \leq \sum_{j \in J} p^i_j - p^i_{j\mu} + (m - r)p^i_{j\mu} \leq m \cdot L^{1}_{\text{OPT}} + (m - r - 1) \cdot L^{1}_{\text{OPT}},
\]

since by the case analysis \( \bar{j}_\mu \) is not scheduled on a machine in \( M^- \). The bounds lead to the same expression as in the previous case, thus concluding the first part of the proof.

A worst-case instance for \( r \leq m - 2 \) machines has the following structure, illustrated in Figure 5.2.1. First, the adversary presents \( r \) huge jobs with processing time \( C > 1 + \frac{r-1}{m-r} \) each. Graham’s algorithm schedules each on a different machine. Next, the adversary presents \( m - r \) big jobs with processing time one. Again, Graham’s algorithm schedules one of these jobs on each of the \( m - r \) idle machines. Then, the adversary presents \((r - 1) \cdot (m - r)\) small jobs with processing time \( \frac{1}{m-r} \). Graham’s algorithm schedules these jobs evenly on the machines with load smaller than \( C \), such that each of these machines has load \( 1 + \frac{r-1}{m-r} \). Finally, a big job with processing time one is presented, and is scheduled on one of the machines with load smaller than \( C \). Now, the perturbation decreases the processing time of all huge jobs from \( C \) to zero. In this way, the online algorithm was forced to work with only \( m - r \) out of the \( m \) available machines, and the achieved makespan is \( 2 + \frac{r-1}{m-r} \). The optimal offline algorithm, on the other hand, schedules each big job on a different machine, thus using \( m - r + 1 \) machines, and schedules \( m - r \) small jobs on each of the remaining \( r - 1 \) machines. In this way, it achieves a makespan of \( 1 \). For \( r = m - 1 \), the bound evaluates to \( L^{1}_{\text{List}} / L^{1}_{\text{OPT}} \leq m \). A simple worst-case example matching this bound first presents \( m - 1 \) big jobs of processing time \( m \), followed by \( m \) small jobs of processing time \( 1 \). The perturbations shrinks the processing time of the \( m - 1 \) big jobs to zero. The stated bound easily follows. The analysis above is not suited for \( r = m \). Nevertheless, the bound \( L^{1}_{\text{List}} / L^{1}_{\text{OPT}} \leq m \) can be achieved with the example for \( r = m - 1 \), by additionally scheduling
Figure 5.2.1: A worst-case example (for \( r = 3 \)) matching the bound of Theorem 5.3. Left, Graham’s schedule on the original instance; the perturbed jobs have a dotted outline. Right, the optimal schedule of the perturbed instance.

a job of processing time \( \frac{1}{m} \) between the big jobs and the small jobs, and by perturbing its processing time to zero as well.

Intuitively, this proof shows that the worst-case scenario happens if the affected machines are blocked with jobs whose processing time decreases to zero. Hence, the adversary and the perturbation force the online algorithm to work with \( r \) machine less than initially stated. Surprisingly, this affects the competitive ratio only with an additive term of \( \frac{r}{m-r} \) with respect to the usual performance of Graham’s algorithm. Intuitively, this increase means that the jobs which can be processed on the affected machines in the optimum offline solution are evenly spread on the unaffected machines in Graham’s algorithm. Finally, for \( r = 0 \) we match the bound for Graham’s algorithm.

5.2.2 Arbitrary increases in processing time

We now consider arbitrary increases in the job’s processing times. The results show that the worst-case perturbation happens if arbitrarily small jobs are scheduled on the machine attaining the makespan before the perturbation, and the perturbation lets these small jobs increase a lot.

Increasing the processing time of one job arbitrarily

**Theorem 5.4.** Restricted to the instances of online scheduling on \( m \) identical parallel machines where the processing time of one job may increase arbitrarily, Graham’s algorithm has a competitive ratio of \( 3 - \frac{2}{m} \), and this bound is best possible.
Proof. We prove the bound by analyzing the whole perturbed schedule up to the makespan. Let $\tilde{j}$ be the job whose processing time was increased, and let $\psi = p_j^\uparrow - p_j$ be the size of the increase. Recall that $\delta_j^k$ is the variation in idle time caused on machine $k$ by the perturbation of job $j$. The following holds:

$$m \cdot L_{\text{List}}^\uparrow = \sum_{j \in J} p_j + \sum_{i \in M} s_i + \psi + \sum_{k \in M} \delta_j^k$$

Now, $-\psi \leq \delta_j^k \leq \psi, k \in M$, since the idle times may increase or decrease as a consequence of the perturbation, and $\psi \leq p_j^\uparrow \leq L_{\text{OPT}}^\uparrow$. Let $a$ be a machine attaining the makespan before the increase, i.e., having $s_a = 0$, and let $b$ be the machine scheduling $\tilde{j}$, which therefore has a decrease in idle time, i.e, $\delta_b^\uparrow \leq 0$. Thus,

$$m \cdot L_{\text{List}}^\uparrow \leq \sum_{j \in J} p_j + \psi + \sum_{i \in M \setminus \{a\}} s_i + \sum_{k \in M \setminus \{b\}} \delta_j^k$$

$$\leq \sum_{j \in J} p_j^\uparrow + (m - 1)L_{\text{OPT}} + (m - 1)L_{\text{OPT}}^\uparrow$$

$$\leq m \cdot L_{\text{OPT}}^\uparrow + (m - 1)L_{\text{OPT}}^\uparrow + (m - 1)L_{\text{OPT}}^\uparrow$$

Hence, we have

$$L(\text{List}(J, P), P^\uparrow) \leq \left(3 - \frac{2}{m}\right) \cdot L(\text{OPT}(J, P^\uparrow), P^\uparrow)$$

A worst-case scenario that comes arbitrarily close to this ratio is as follows, and is illustrated in Figure 5.2.2. Consider an adversary that starts by sequentially feeding the online algorithm with $m - 2$ jobs with processing time $1 - \frac{2}{m}$. Graham’s algorithm schedules each of these jobs on a different machine. Next, the adversary sequentially present $2m - 5$ jobs with processing time $\frac{1}{m}$, followed by a job with processing time $\frac{1}{m} - \varepsilon$, for an arbitrarily small $\varepsilon \in \mathbb{Q}_0^+, 0 < \varepsilon < \frac{1}{m}$. Graham’s algorithm schedules these jobs alternating between the two machines with a load smaller than one. At the end of this process, all but one machine have load $1 - \frac{2}{m}$, and one machine has load $1 - \frac{2}{m} - \varepsilon$. Finally, the adversary presents a job of processing time $\varepsilon^2$ and a last job of processing time $1$. Both jobs are scheduled in this order on the machine with load $1 - \frac{2}{m} - \varepsilon$. Now, a worst-case
perturbation increases the job of processing time $\varepsilon^2$ to 1, yielding a makespan of $3 - \frac{2}{m} - \varepsilon$. The optimal offline algorithm, on the other hand, schedules the two jobs with processing time one each on a single machine, and schedules one job with processing time $1 - \frac{2}{m}$ and two jobs of processing time at most $\frac{1}{m}$ on each of the remaining $m - 2$ machines. This schedule leads to a makespan of 1. Hence, we have a competitive ratio of $3 - \frac{2}{m} - \varepsilon$, which comes arbitrarily close to the upper bound.

### Increasing the processing time of $r$ jobs arbitrarily

The previous theorem can be generalized to the case where $r$ jobs increase their processing time arbitrarily. Then, we give a similar analysis for the case where any number of jobs may be perturbed, and the jobs result to be scheduled on $r$ machines of an optimal offline schedule of the perturbed instance. Also in this case, our analysis matches Graham’s bound for $r = 0$.

**Theorem 5.5.** Consider the instances of online scheduling on $m$ identical parallel machines where perturbations may increase the processing times of $r$ jobs arbitrarily. Restricted to these instances, Graham’s algorithm has a competitive ratio of $2 + r - \frac{r+1}{m}$, and this bound is best possible for $r \leq m - 2$. 
5.2. Arbitrary perturbations

Proof. Let $J^\uparrow \subset J$, $|J^\uparrow| = r$ be the arbitrarily ordered list of jobs whose processing times increase. Let $\psi_j = p_j^\uparrow - p_j$ be the increase of job $j \in J^\uparrow$, which can be bounded by $\psi_j \leq p_j^\uparrow \leq L_{\OPT}^\uparrow$, $j \in J^\uparrow$. Recall that for the analysis we assume the perturbations to occur sequentially. Thus, each perturbed job has an associated difference in idle time for each machine; hence $-\psi_j \leq \delta_{kj} \leq \psi_j$, $k \in M$, $j \in J^\uparrow$. Furthermore, at least one machine $\mu \in M$ attains the makespan before the perturbation, and has idle time $s_\mu = 0$. The idle times of the remaining machines satisfy $s_i \leq L_{\OPT}^\uparrow \leq L_{\OPT}^\uparrow$, $i \in M$. Furthermore, the increase of job $j \in J^\uparrow$ does not increase the idle time of the machine the job is scheduled on (although it may decrease it). The new makespan of the machines is thus given by:

$$m \cdot L_{\text{List}}^\uparrow = \sum_{j \in J} p_j + \sum_{i \in M} s_i + \sum_{j \in J^\uparrow} \psi_j + \sum_{j \in J^\uparrow, k \in M} \delta_{kj}$$

$$\leq \sum_{j \in J} p_j^\uparrow + \sum_{i \in M} s_i + \sum_{j \in J^\uparrow, k \in M} \delta_{kj}$$

$$\leq m L_{\OPT}^\uparrow + (m - 1) L_{\OPT}^\uparrow + r (m - 1) L_{\OPT}^\uparrow,$$

$L(\text{List}(J, P), P^\uparrow) \leq \left(2 + r - \frac{r + 1}{m}\right) \cdot L(\text{OPT}(J, P^\uparrow), P^\uparrow)$.

To see that the analysis is best possible for $r \leq m - 2$, we give an example, illustrated in Figure 5.2.3, where this bound is achieved up to an arbitrarily small $\varepsilon \in \mathbb{Q}_0^+$, $0 < \varepsilon < \frac{1}{m}$. The adversary uses three types of jobs, namely big, small and tiny jobs. The processing times of the jobs within each class are similar and specified in the following. First, the adversary presents $m - r - 1$ big jobs with processing time $1 - \frac{(r + 1)}{m}$. Graham’s algorithm schedules each on a different machine. The adversary presents $m(r + 1)(1 - \frac{r + 1}{m}) - 1 = (r + 1)(m - r - 1) - 1$ small jobs with processing time $\frac{1}{m} - \varepsilon$, for $\varepsilon$ arbitrarily small. Graham’s algorithm schedules the jobs evenly on the machines processing no big jobs. At this point, all but one machine have a load of $1 - \frac{r + 1}{m}$, and one machine has load $1 - \frac{r + 1}{m} - \varepsilon$. Now, the adversary presents $r$ tiny jobs with processing time $\frac{\varepsilon}{r} - \varepsilon^2$ and a last job with processing time 1. The perturbation increases the processing times of tiny jobs from $\frac{\varepsilon}{r} - \varepsilon^2$ to 1. The makespan of Graham’s schedule is $2 + r - \frac{r + 1}{m} - \varepsilon$. On the other hand, the optimal offline algorithm on the perturbed instance schedules each of the now $m$ big jobs on
Theorem 5.5 can be generalized as follows. Let $J_i^\uparrow$ be the arbitrarily ordered list of perturbed jobs which are scheduled on machine $i \in M$ in an optimum offline solution $OPT(J, P^\uparrow)$. Let $M^\neq = \{i \in M|J_i^\uparrow \neq \emptyset\}$ be the set of machines in the considered optimal offline solution which process at least one job with perturbed processing time, and define $r = |M^\neq|$. Note that here the perturbed machines are defined with respect to $OPT(J, P^\uparrow)$, and not to Graham’s schedule.

Theorem 5.6. Consider the instances of online scheduling on $m$ identical parallel machines with the following properties: first, the processing times of some jobs may increase arbitrarily; second, the perturbed jobs are scheduled on $r$ machines of an optimal offline perturbed solution. For these instances, Graham’s algorithm has a competitive ratio of $2 + r - \frac{r+1}{m}$, and this bound is best possible for $r \leq m - 2$.

Proof. Let $J^\uparrow = J_1^\uparrow J_2^\uparrow \cdots J_m^\uparrow$ be the concatenation of the ordered lists $J_i^\uparrow$. Let $\psi_j = p^\uparrow_j - p_j, j \in J^\uparrow$ be the increase in processing time of the perturbed jobs. For the analysis of Graham’s schedule, we may assume that the increases in processing time occur sequentially in the
5.2. Arbitrary perturbations

order given in the lists $J \uparrow$. Therefore, each job $j \in J \uparrow$ causes a well defined variation $\delta^k_j$ in idle time on each machine $k \in M$ of Graham’s schedule. This variation is bounded by $-\psi_j \leq \delta^k_j \leq \psi_j$; furthermore, this variation is negative or zero on the machine where $j \in J \uparrow$ is scheduled by Graham’s algorithm. Thus, $\sum_{k \in M} \delta^k_j \leq (m - 1) \cdot \psi_j, j \in J \uparrow$. Finally, $\sum_{j \in J \uparrow} \psi_j \leq \sum_{j \in J \uparrow} p^\uparrow_j \leq \mathcal{L}_{\text{OPT}}^\uparrow, i \in M$, since all the jobs in $J_i \uparrow$ are scheduled on the same machine in the considered optimal offline schedule of the perturbed instance. Therefore,

$$m \cdot \mathcal{L}_{\text{List}}^\uparrow = \sum_{j \in J} p_j + \sum_{i \in M} s_i + \sum_{j \in J \uparrow} \psi_j + \sum_{j \in J \uparrow} \sum_{k \in M} \delta^k_j,$$

and we can bound the increases in idle time as follows:

$$\sum_{j \in J \uparrow} \sum_{k \in M} \delta^k_j = \sum_{i \in M} \sum_{j \in J_i \uparrow} \sum_{k \in M} \delta^k_j \leq \sum_{i \in M} \sum_{j \in J_i \uparrow} (m - 1) \psi_j \leq (m - 1) \sum_{i \in M} \mathcal{L}_{\text{OPT}}^\uparrow \leq (m - 1) \cdot r \cdot \mathcal{L}_{\text{OPT}}^\uparrow.$$

Hence, we have

$$m \cdot \mathcal{L}_{\text{List}}^\uparrow \leq \sum_{j \in J} p_j + \sum_{i \in M} s_i + \sum_{j \in J \uparrow} \psi_j + r(m - 1)\mathcal{L}_{\text{OPT}}^\uparrow \leq \sum_{j \in J} p^\uparrow_j + \sum_{i \in M} s_i + r(m - 1)\mathcal{L}_{\text{OPT}}^\uparrow \leq m \cdot \mathcal{L}_{\text{OPT}}^\uparrow + (m - 1) \cdot \mathcal{L}_{\text{OPT}}^\uparrow + r(m - 1)\mathcal{L}_{\text{OPT}}^\uparrow,$$

since at least one machine attained the makespan before the perturbation and had zero idle time. Thus, we have the following bound on the makespan:

$$\mathcal{L}(\text{List}(J, P), P^\uparrow) \leq \left(2 + r - \frac{r + 1}{m}\right) \cdot \mathcal{L}(\text{OPT}(J, P^\uparrow), P^\uparrow).$$

The bound is best-possible for $r \leq m - 2$, since the example of Theorem 5.5 affects $r$ machines. $\square$
5.3 Bounded perturbations

In the following analyses, we constrain the effect perturbations may have. Here, we assume that the perturbed processing time is related to the original processing time, and bound the perturbed processing times by a constant factor of their original processing time. Thus, the perturbed processing times are of the form $\tilde{p} = \alpha p$ for some $\alpha$. Values of $\alpha > 1$ imply perturbations increasing the processing times, whereas $\alpha < 1$ imply perturbations decreasing processing times. In this section, we analyze both cases separately.

5.3.1 Bounded decreases in processing times

In this section, we analyze the behavior of the makespan given that the processing times of jobs may decrease by a bounded factor. We start by stating a simple lower bound.

**Theorem 5.7.** Consider the instances of online scheduling on $m \geq 2$ identical parallel machines where the processing time of at most one job may decrease to a factor at least $\frac{1}{x}$ of its original processing time, for $x > 1, x \in \mathbb{Q}_0^+$. No online algorithm applied to these instances can achieve a competitive ratio smaller than $2 - \frac{2}{x+1}$.

The stated bound is $\frac{4}{3}$ for halving processing times, and is arbitrarily close to the lower bound of 2 stated in Theorem 5.1 for arbitrarily large $x$.

**Proof.** Consider an adversarial sequence of $m + 1$ jobs of processing time 1. Any algorithm aiming at a competitive ratio strictly smaller than 2 must schedule the first $m$ jobs each on a different machine. Were this not so, the adversary could stop the sequence after the first job has been scheduled on a machine with nonzero load, thus enforcing a competitive ratio of two. The last job may be scheduled on any machine. Now, the perturbation decreases the processing time of a job which is scheduled alone on a machine from 1 to $\frac{1}{x}$. Any online algorithm aiming at a competitive ratio smaller than two has a makespan of two, whereas the optimum offline perturbed schedule achieves a makespan of $1 + \frac{1}{x}$ by scheduling a job of processing time 1 on the machine where the perturbed job is scheduled, and by scheduling the
remaining \( m - 1 \) jobs each on one machine. The lower bound follows from the ratio of the two makespans.

A lower bound of 2 on the competitive ratio can be shown for the case where at most 2 jobs at most halve their processing time due to the perturbations:

**Theorem 5.8.** Consider the instances of online scheduling on \( m \geq 3 \) identical parallel machines where the processing time of at most two jobs may decrease to a factor at least \( \frac{1}{2} \) of their original processing time. Applied to these instances, no online algorithm can have a competitive ratio of less than 2.

**Proof.** By contradiction. Assume an online algorithm with a strictly smaller competitive ratio exists, and consider an adversarial sequence of \( m + 1 \) jobs of processing time 1. By the same argument as in the proof of Theorem 5.7, any online algorithm aiming at a competitive ratio strictly smaller than 2 has a makespan of two. The perturbation decreases the processing times of two jobs that are scheduled alone each on one machine to \( \frac{1}{2} \), which leaves the makespan of the online algorithm unchanged. The optimum offline perturbed schedule achieves a makespan of 1 by scheduling the two perturbed jobs on the same machine and the remaining \( m - 1 \) jobs each on a machine. The ratio of the two objectives is two, a contradiction to our assumption.

Having assessed these lower bounds, we analyze the quality of the schedules built by Graham’s algorithm when facing these kinds of perturbations.

**Theorem 5.9.** Consider the instances of online scheduling on \( m \) identical parallel machines where the processing times of an arbitrary number of jobs may decrease to a factor at least \( \frac{1}{x} \) of their original processing time, for \( x > 1, x \in \mathbb{Q}_0^+ \). Restricted to these instances, Graham’s algorithm has a competitive ratio between \( 1 + x - \frac{x^2}{m-1+x} \) and \( 1 + x - \frac{x}{m} \), for a small \( \varepsilon \in \mathbb{Q}_0^+ \), \( 0 < \varepsilon < \frac{x}{m-1+x} \).

**Proof.** Consider a machine \( \mu \) attaining the makespan after the perturbation. Let \( j_\mu \) be the last job scheduled on \( \mu \), and let \( \ell \) be the load of \( \mu \) on the original instance when \( j_\mu \) was presented. Hence, the total load of \( \mu \) before the perturbation is \( \ell + p_{j_\mu} \). Because \( j_\mu \) was
Chapter 5. Robust Online Scheduling

scheduled on \( \mu \), all machines have at least load \( \ell \) with the original processing times. Therefore, \( m \ell \leq \sum_{j \in J \setminus \{j_\mu\}} p_j \). Since at most all jobs are perturbed and decrease to a factor at least \( \frac{1}{x} \), \( \sum_{j \in J \setminus \{j_\mu\}} p_j \leq x \sum_{j \in J \setminus \{j_\mu\}} p_j \leq x m \cdot L_{\text{OPT}} \) where \( x \) is a factor. Hence, \( \ell \leq x L_{\text{OPT}} - x \cdot p_{j_\mu} \), since at most all jobs are perturbed and decrease to a factor at least \( \frac{1}{x} \).

Thus:

\[
\mathcal{L}(\text{List}(J, P), P^\downarrow) \leq \ell + p_{j_\mu} \leq x \cdot L_{\text{OPT}} - x \cdot \frac{p_{j_\mu}}{m} + p_{j_\mu}
\]

\[
\leq x \cdot L_{\text{OPT}} + \left( 1 - \frac{x}{m} \right) p_{j_\mu}
\]

\[
\leq \left( 1 + x - \frac{x}{m} \right) \mathcal{L}(\text{OPT}(J, P^\downarrow), P^\downarrow).
\]

An example which comes close to this upper bound is the following sequence of jobs, illustrated in Figure 5.3.1 for the case \( x = 2 \). The adversary presents \( m - 1 \) big jobs with processing time \( x - \frac{x^2}{m-1+x} \), \( m - 2 \) small jobs with processing time \( \frac{x}{m-1} \left( 1 - \frac{x}{m-1+x} \right) \), one small job with processing time \( \frac{x}{m-1} \left( 1 - \frac{x}{m-1+x} \right) - \varepsilon \), for a small \( \varepsilon \in \mathbb{Q}^+ \), \( 0 < \varepsilon < \frac{x}{m-1} \left( 1 - \frac{x}{m-1+x} \right) \), followed by a last job with processing time 1. The perturbation affects the \( m - 1 \) big jobs, decreasing their processing time to \( 1 - \frac{x}{m-1+x} \). Graham’s schedule on the perturbed instance has a makespan of \( 1 + x - \frac{x^2}{m-1+x} - \varepsilon \), whereas the optimum offline perturbed solution has a makespan \( \mathcal{L}(\text{OPT}(J, P^\downarrow), P^\downarrow) = 1 \).

\[\square\]

**Corollary 5.10.** Consider the instances of online scheduling on \( m \) identical parallel machines where the processing times of an arbitrary number of jobs may halve. Restricted to these instances, Graham’s algorithm has a competitive ratio of at most \( 3 - \frac{4}{m+1} - \varepsilon \), for \( \varepsilon \in \mathbb{Q}^+ \), \( 0 < \varepsilon < \frac{2}{m-1} - \frac{4}{m^2-1}, m > 1 \).

The corollary follows by setting \( x = 2 \) in Theorem 5.9. By setting \( x = 1 \) we match the bound for Graham’s algorithm. In the following, we bound the number of jobs which a perturbation may decrease. We show the following Theorem:

**Theorem 5.11.** Consider the instances of online scheduling on \( m \) identical parallel machines where the perturbations may decrease the processing times of \( r \) jobs to a factor at least \( \frac{1}{x} \) of their original processing time, for \( x \in \mathbb{Q}^+, x > 1 \). Restricted to these instances, Graham’s algorithm has a competitive ratio of \( 2 + \frac{r \cdot (x-1)-1}{m} \).
5.3. Bounded perturbations

\[ m - \frac{x - x^2}{(m - 1 + x)} \]

\[ m - \frac{x}{m - 1 + x} \]

\[ m - \frac{1 - x}{(m - 1 + x)} \]

\[ m - \frac{1 - \frac{x}{m - 1 + x}}{} - \varepsilon \]

\[ m - \frac{1 - \frac{x}{m - 1 + x}}{} - \varepsilon \]

Figure 5.3.1: The bad example of Theorem 5.9 for \( x = 2 \). Top left, the Graham’s schedule on the original instance. Bottom left, the Graham’s schedule after the perturbation. Right, the optimum offline perturbed schedule.

Proof. Consider a machine \( \mu \) attaining the makespan after the perturbation. Let \( \bar{j}_\mu \) be the last job scheduled on \( \mu \), and let \( \ell \) be the load of \( \mu \) in the original instance when \( \bar{j}_\mu \) was presented. Therefore, \( m\ell \leq \sum_{j \in J \setminus \{\bar{j}_\mu\}} p_j \). Let \( J_r \) be the set of the \( r \) perturbed jobs with the exception of \( \bar{j}_\mu \), given that \( \bar{j}_\mu \) was perturbed. Then, we have:

\[
m\ell \leq \sum_{j \in J \setminus \{\bar{j}_\mu\} \setminus J_r} p_j + \sum_{j \in J_r} p_j \leq \sum_{j \in J \setminus \{\bar{j}_\mu\} \setminus J_r} p_j^\downarrow + x \cdot \sum_{j \in J_r} p_j^\downarrow
\]

\[
= \sum_{j \in J} p_j^\downarrow + (x - 1) \sum_{j \in J_r} p_j^\downarrow - p_{\bar{j}_\mu}^\downarrow
\]

\[
\leq (m + (x - 1) \cdot r) \mathcal{L}_\text{OPT} - p_{\bar{j}_\mu}^\downarrow.
\]

since \( p_j^\downarrow \leq \mathcal{L}_\text{OPT}, j \in J_r \). Finally, as the makespan of Graham’s algorithm can be bounded by \( \mathcal{L}_\text{List} \leq \ell + p_{\bar{j}_\mu}^\downarrow \), we have:

\[
\mathcal{L}(\text{List}(J, P), P^\downarrow) \leq \left( 2 + \frac{(x - 1) \cdot r - 1}{m} \right) \mathcal{L}(\text{OPT}(J, P^\downarrow), P^\downarrow).
\]

\[ \square \]
Corollary 5.12. Graham’s algorithm for online scheduling on \( m \) identical parallel machines, applied to instances where the processing time of one job may halve, has a competitive ratio not greater than 2 and no smaller than 2 \(- \frac{1}{2m-1} - \varepsilon \), for an arbitrary small \( \varepsilon \in \mathbb{Q}_0^+, 0 < \varepsilon < \frac{1}{2m-1} \).

Proof. For the upper bound, it is sufficient to set \( x = 2 \) and \( r = 1 \) in Theorem 5.11. A bad example achieving the lower bound is the following sequence of jobs: the adversary presents \( m - 1 \) big jobs with processing time \( 1 - \frac{1}{2m-1} \), which Graham’s algorithm schedules each on a different machine. Then, \( 2(m - 1) - 1 \) small jobs with processing time \( \frac{1}{2m-1} \), and a small job with processing time \( \frac{1}{2m-1} - \varepsilon, \varepsilon \in \mathbb{Q}_0^+, 0 < \varepsilon < \frac{1}{2m-1} \) follow. All small jobs are scheduled on the same initially empty machine. Finally, the adversary presents a job with processing time 1, which is scheduled together with the small jobs. Now, the perturbation affects any big job, decreasing its processing time to \( \frac{1}{2} - \frac{1}{2} \frac{1}{2m-1} \). The makespan of this schedule is \( 2 - \frac{1}{2m-1} - \varepsilon \). The optimum offline perturbed scheduled has a makespan of 1: it schedules each of the \( m - 2 \) jobs with processing time \( 1 - \frac{1}{2m-1} \) together with one job of processing time \( \frac{1}{2m-1} \) on a separate machine, the job of processing time one alone on one machine, and the remaining \( m \) small jobs with the job of processing time \( \frac{1}{2} - \frac{1}{2} \frac{1}{2m-1} \) on the remaining machine. \( \square \)

Similar to Theorem 5.6, Theorem 5.9 can also be stated with respect to the number \( r \) of affected machines \( M \neq \emptyset \) in an optimum offline solution \( \text{OPT}(J, P^\downarrow) \).

Theorem 5.13. Consider the instances for online scheduling on \( m \) identical parallel machines with the following two properties: first, the perturbations may decrease the processing times of some jobs to a factor of at least \( \frac{1}{x} \) of their original processing time, for \( x \in \mathbb{Q}_0^+, x > 1 \); second, the perturbed jobs are scheduled on \( r \) machines in an optimum offline solution. Restricted to these instances, Graham’s algorithm has a competitive ratio of \( 2 + \frac{r \cdot (x - 1) - 1}{m} \).

Proof. Let \( J_i^\downarrow \) be the set of perturbed jobs scheduled on machine \( i \in M \) in \( \text{OPT}(J, P^\downarrow) \), and let \( M \neq \emptyset = \{ i \in M | J_i^\downarrow \neq \emptyset \}, r = |M \neq \emptyset | \). Let \( \bar{J} = \bigcup_{i \in M \neq \emptyset} J_i^\downarrow \) be the set of all perturbed jobs. Consider a machine \( \mu \) attaining the makespan after the perturbation, let \( \bar{j}_\mu \) be the last job
5.3. Bounded perturbations

scheduled on $\mu$, and let $\ell$ be the load of $\mu$ in the original instance when $\bar{J}_\mu$ was presented. Therefore, $L(\text{List}(J, P), P^\perp) \leq \ell + p_{\bar{J}_\mu}$. Now:

$$m\ell \leq \sum_{j \in J \setminus \bar{J}_\mu} p_j \leq \sum_{j \in \bar{J}} p_j + \sum_{j \in J \setminus \bar{J}} p_j - p_{\bar{J}_\mu}$$

$$\leq x \sum_{j \in \bar{J}} p_j^\perp + \sum_{j \in J \setminus \bar{J}} p_j^\perp - p_{\bar{J}_\mu} \leq \sum_{j \in J} p_j^\perp + (x - 1) \sum_{j \in \bar{J}} p_j^\perp - p_{\bar{J}_\mu}$$

$$\leq \sum_{j \in J} p_j^\perp + (x - 1) \sum_{i \in M^\neq \Bar{J}_i} \sum_{j \in J_i^\perp} p_j^\perp - p_{\bar{J}_\mu}$$

$$\leq mL^\perp_{\text{OPT}} + (x - 1) rL^\perp_{\text{OPT}} - p_{\bar{J}_\mu}$$

$$= L^\perp_{\text{OPT}} (m + (x - 1) r) - p_{\bar{J}_\mu}.$$  

where the last inequality follows because $\sum_{j \in J_i^\perp} p_j^\perp \leq L^\perp_{\text{OPT}}$. Thus,

$$L^\perp_{\text{List}} \leq L^\perp_{\text{OPT}} + \frac{(x - 1) r}{m} L^\perp_{\text{OPT}} + \left(1 - \frac{1}{m}\right) p_{\bar{J}_\mu}$$

$$\leq \left(2 + \frac{(x - 1) r - 1}{m}\right) L^\perp_{\text{OPT}}.$$

\[
\]

5.3.2 Bounded increases in processing times

We start by analyzing the impact of doubling the processing time of one job, proceed by considering the increase by a constant factor of one job’s processing time, then extend the analysis to allowing many jobs to change in processing time by a bounded amount.

Surprisingly, we shall see that the last analysis suggests that it’s not that relevant how many jobs we are allowed to increase in processing time, but that if we allow an $x$-factor increase in processing time, $x$ perturbed jobs are sufficient to produce a worst-case behavior.

Bounded increase of one job

In this section, we first show a simple lower bound for any online algorithm on instances where the processing time of one job may
double; then, we show that this setting does not affect Graham’s algorithm as badly as the arbitrary increase of one job.

**Theorem 5.14.** No algorithm for online scheduling on \( m \geq 2 \) identical parallel machines can have a competitive ratio strictly smaller than \( \frac{3}{2} \) if the processing time of one job may double.

**Proof.** Assume this was possible and consider an online algorithm which achieves a competitive ratio better than \( \frac{3}{2} \). The adversarial sequence consists of \( m + 1 \) jobs with processing time 1. To achieve a competitive ratio smaller than two, the online algorithm schedules the first \( m \) jobs each on a different machine. The last job may be scheduled on any machine. The perturbation now doubles the processing time of the job scheduled last. By doing so, this schedule has a makespan of 3, whereas the optimal offline perturbed schedule has a makespan of 2, obtained by scheduling the only job with processing time 2 alone on a machine, and by scheduling the other jobs in such a way that the maximum load of any machine is two.

We now analyze the performance of Graham’s algorithm in this situation.

**Theorem 5.15.** Graham’s algorithm for online scheduling on \( m \) identical parallel machines has a competitive ratio of \( \left( 2.5 - \frac{3}{2m} \right) \) on instances where the processing time of at most one job doubles, and this bound is best possible for \( m \geq 3 \).

Hence, Graham’s algorithm has an offset close to 1 with respect to the lower bound.

**Proof.** Let \( j \) be the perturbed job and \( \mu \) the machine where \( j \) is scheduled, \( a \) the machine attaining the makespan in the original instance. The processing time of the machines is expressed as follows:

\[
\begin{align*}
    m \cdot L_{\text{List}}^\uparrow &= \sum_{j \in J} p_j^\uparrow + \sum_{i \in M} s_i^\uparrow \leq \sum_{j \in J} p_j^\uparrow + \sum_{i \in M \setminus \{a\}} s_i + \sum_{i \in M \setminus \{\mu\}} p_i^\uparrow \left( \frac{1}{2} \right) \\
    &\leq m \cdot L_{\text{OPT}}^\uparrow + (m - 1) \cdot \left( L_{\text{OPT}}^\uparrow + \frac{L_{\text{OPT}}^\uparrow}{2} \right),
\end{align*}
\]
The inequalities hold since the increase of each idle time is at most the size of the increase of \( \tilde{j} \)'s processing time, which is half of its perturbed processing time. Furthermore, \( \tilde{j} \)'s perturbed processing time is bounded by \( L_{\text{OPT}}^\uparrow \) and the machine where \( \tilde{j} \) is scheduled on does not have an increase in idle time. Hence,

\[
L(\text{List}(J, P), P^\uparrow) \leq \left(2.5 - \frac{3}{2m}\right) \cdot L(\text{OPT}(J, P^\uparrow), P^\uparrow).
\]

The bound is best possible, as can be shown with the following simple instance, see Figure 5.3.2. First, the adversary presents \((m - 3)\) jobs with processing time \(1 - \frac{3}{2m}\). These are scheduled each on a different machine. Next, the adversary presents one job \(j_a\) with processing time \(\frac{1}{2} - \frac{3}{2m} - \varepsilon\), for an arbitrarily small \(\varepsilon \in \mathbb{Q}_0^+, \frac{1}{2m} > \varepsilon > 0\). Again, this job is scheduled on an empty machine. Then, the adversary presents \(2m - 6\) small jobs with processing time \(\frac{1}{2m}\). These small jobs are evenly scheduled on the two machines which were idle up to that point, enforcing a load of \(\frac{1}{2} - \frac{3}{2m}\) on each of the two machines. Next, the adversary presents one job \(j_b\) with processing time \(\frac{1}{2}\), which is scheduled on the machine processing \(j_a\). Then, it’s the turn of \(2m\) small jobs with processing time \(\frac{1}{2m}\) each, which are scheduled evenly on the two machines already processing small jobs. Finally, the adversary presents one job with processing time \(1\), which is scheduled on the machine processing \(j_a\) and \(j_b\). Now, the perturbation increases the processing time of job \(j_b\) from \(\frac{1}{2}\) to \(1\), thus giving a makespan of \(2.5 - \frac{3}{2m} - \varepsilon\). The optimum offline algorithm, on the other hand, achieves a makespan of 1 by scheduling the two jobs with processing time \(1\) on a machine each, 3 jobs with processing time \(\frac{1}{2m}\) and one job with processing time \(1 - \frac{3}{2m}\) on each of \(m - 3\) machines, and the job with processing time \(\frac{1}{2} - \frac{3}{2m} - \varepsilon\) with \(m + 3\) jobs with processing time \(\frac{1}{2m}\) on the last machine.

Note that if the makespan is not increased by the perturbation, the competitive ratio of \(2 - \frac{1}{m}\) is maintained.

We show next that increasing one job’s processing time to a factor \(x > 1\) gives us a swift transition from the case of bounded perturbations to arbitrary perturbations.

**Theorem 5.16.** Graham’s algorithm for online scheduling on \(m\) identical parallel machines on instances where the processing time of one job is perturbed to a factor \(x > 1\), \(x \in \mathbb{Q}_0^+\) of its original processing
time has a competitive ratio of \(3 - \frac{1}{x} - \frac{(2 - \frac{1}{x})}{m}\). This bound is best possible for \(m \geq 3\) and \(x \geq \frac{m-1}{m-2}\).

Proof. Let \(\tilde{j}\) be the perturbed job. In this case, \(p_j^\uparrow = x \cdot p_j\). Therefore, the idle time of each machine may increase by at most the amount of this increase, that is, by \(\frac{x-1}{x}p_j^\uparrow\), and the machine \(\mu\) where \(\tilde{j}\) is scheduled has no increase. Naturally, since the optimal offline schedule needs to process \(\tilde{j}\) as well, \(p_j^\uparrow \leq \mathcal{L}^\uparrow_{\text{OPT}}\). Finally, since the processing times may only increase, the idle times of the original schedule are bounded by \(\mathcal{L}^\uparrow_{\text{OPT}}\), and the machine \(a\) which attains the makespan \(\mathcal{L}(\text{List}(J, P), P)\) has zero idle time. Now, the schedule of the machines up to the makespan \(\mathcal{L}(\text{List}(J, P), P^\uparrow)\) is described as follows:

\[
m \cdot \mathcal{L}_{\text{List}}^\uparrow = \sum_{j \in J} p_j^\uparrow + \sum_{i \in M} s_i^\uparrow \leq \sum_{j \in J} p_j^\uparrow + \sum_{i \in M \setminus \{a\}} s_i + \sum_{i \in M \setminus \{\mu\}} \frac{x-1}{x} p_j^\uparrow
\]

\[
\leq m \cdot \mathcal{L}^\uparrow_{\text{OPT}} + (m - 1) \cdot \left( \mathcal{L}^\uparrow_{\text{OPT}} + \frac{x-1}{x} \mathcal{L}^\uparrow_{\text{OPT}} \right).
\]
Thus,
\[ L(\text{List}(J, P), P^\uparrow) \leq \left( 3 - \frac{1}{x} - \frac{(2 - \frac{1}{x})}{m} \right) L(\text{OPT}(J, P^\uparrow), P^\uparrow). \]

An example coming arbitrarily close to this bound for \( x = \frac{p}{q} \in \mathbb{Q}_0^+ \) follows and is a generalization of the example for Theorem 5.15. Note that the restriction \( m \geq 3 \) is necessary since at least 3 machines are needed to schedule the jobs with a makespan of 1 for the optimum offline perturbed schedule, and \( x \geq \frac{m-1}{m-2} \) is required for ensuring that all processing times are positive. The adversary sequentially presents \( m - 3 \) jobs with processing time \( 1 - \frac{2-\frac{1}{x}}{m} \), one job with processing time \( 1 - \frac{1}{x} - \frac{2-\frac{1}{x}}{m} - \varepsilon \), \( 2pm - 2qm - 4p + 2q \) jobs with processing time \( \frac{1}{pm} \), one job with processing time \( \frac{1}{x} \), \( 2qm \) jobs with processing time \( \frac{1}{pm} \) and finally a job with processing time 1. The perturbation increases the job with processing time \( \frac{1}{x} \) to 1, leading to a makespan of \( \left( 3 - \frac{1}{x} - \frac{(2 - \frac{1}{x})}{m} \right) \), whereas the optimum offline algorithm on the perturbed instance achieves a makespan of 1.

Note that, as expected, for \( x = 2 \) we match the result of Theorem 5.15; furthermore, for big \( x \) we get arbitrarily close to the result of Theorem 5.4. Finally, with \( x = 1 \) we match the bound of Graham’s algorithm.

**Bounded increase of many jobs**

We again begin by showing a simple lower bound.

**Theorem 5.17.** No online algorithm for online scheduling on \( m \) identical parallel machines can have a competitive ratio strictly smaller than 2 if the processing time of two jobs may double.

**Proof.** The proof is similar to the proof of Theorem 5.14. Assume an online algorithm with a competitive ratio strictly smaller than two exists. The adversarial sequence for this algorithm consists of \( m + 1 \) jobs of processing time 1. By the same arguments of Theorem 5.7, no two jobs within the first \( m \) can be scheduled on the same machine by the online algorithm. The last of the jobs of the sequence can be
scheduled on any machine. Consider the jobs scheduled on the machine having a load of two. The worst-case perturbation doubles the processing time of these jobs, enforcing a makespan of 4. The optimum offline perturbed schedule, on the other hand, schedules the two jobs of processing time two each on a single machine, and distributes the remaining \( m - 1 \) jobs of processing time 1 on \( m - 2 \) machines in such a way that no machine exceeds a load of 2. The ratio of the two makespans proves a competitive ratio of 2, a contradiction.

The previous analyses of Graham’s algorithm can be extended to the cases where the perturbation affects more than one job. Our analysis shows that the actual number of perturbed jobs is not really relevant, but is tied to the amount of the perturbation. Here, we show that perturbing two jobs is sufficient to achieve a worst-case behavior of Graham’s algorithm.

**Theorem 5.18.** Graham’s algorithm for online scheduling on \( m \) identical parallel machines on instances where the processing time of many jobs doubles has a competitive ratio of at most \( 3 - \frac{1}{m} \) and of at least \( 3 - \frac{4}{m + 1} \), for an arbitrarily small \( \varepsilon < \frac{1}{2(m-1)} \), \( \varepsilon \in \mathbb{Q}_0^{+} \) and \( m \geq 3 \).

**Proof.** This proof follows a slightly different approach than the previous ones. Let \( \mu \) be a machine which attains the makespan \( L_{\text{List}}^{\uparrow} \).

Consider the jobs scheduled on \( \mu \) in the state before the perturbation. We can subdivide the processing times on \( \mu \) into three disjoint parts: \( \alpha \), the processing time of the last scheduled job on the machine; \( \beta \), the total processing time of jobs which double, excluding \( \alpha \) if the last job is perturbed; \( \gamma \), the processing time of the jobs that do not double, again excluding \( \alpha \) if the job is not perturbed. Because of Graham’s algorithm, we know that before the perturbation \( \beta + \gamma \leq L_{\text{OPT}}^{\downarrow} - \alpha \), since all machines have load at least \( \beta + \gamma \) when the job with processing time \( \alpha \) is scheduled. Hence, \( \beta \leq L_{\text{OPT}}^{\downarrow} - \frac{\alpha}{m} - \gamma \) follows.

Because the processing times only increase, \( L_{\text{OPT}}^{\uparrow} \leq L_{\text{OPT}}^{\downarrow} \). Furthermore, let \( \alpha^{\uparrow} \) and \( \beta^{\uparrow} \) be the perturbed counterparts of \( \alpha \) and \( \beta \). After the doubling, the makespan is:

\[
L_{\text{List}}^{\uparrow} = \gamma + \beta^{\uparrow} + \alpha^{\uparrow} = \gamma + 2\beta + \alpha^{\uparrow}.
\]

We now distinguish two cases:
5.3. Bounded perturbations

\( \alpha \) doubles: in this case, \( 2\alpha = \alpha^\uparrow \leq \mathcal{L}_{\text{OPT}}^\uparrow \); the makespan is:

\[
\mathcal{L}_{\text{List}} = \gamma + 2\alpha + 2\beta \leq \gamma + 2\alpha + 2\mathcal{L}_{\text{OPT}} - 2 \frac{\alpha}{m} - 2\gamma
\]

\[
\leq \mathcal{L}_{\text{OPT}}^\uparrow \left( 1 - \frac{1}{m} \right) + 2\mathcal{L}_{\text{OPT}}^\uparrow - \gamma \leq \left( 3 - \frac{1}{m} \right) \mathcal{L}_{\text{OPT}}^\uparrow .
\]

\( \alpha \) does not double: then, \( \alpha \leq \mathcal{L}_{\text{OPT}} \leq \mathcal{L}_{\text{OPT}}^\uparrow \), and the makespan is:

\[
\mathcal{L}_{\text{List}} = \gamma + 2\beta + \alpha \leq \gamma + 2\mathcal{L}_{\text{OPT}} - 2 \frac{\alpha}{m} - 2\gamma + \alpha
\]

\[
\leq 2\mathcal{L}_{\text{OPT}}^\uparrow + \left( 1 - \frac{2}{m} \right) \mathcal{L}_{\text{OPT}}^\uparrow - \gamma \leq \left( 3 - \frac{2}{m} \right) \mathcal{L}_{\text{OPT}}^\uparrow .
\]

Therefore, \( \mathcal{L}(\text{List}(J, P), P^\uparrow) \leq \left( 3 - \frac{1}{m} \right) \mathcal{L}(\text{OPT}(J, P^\uparrow), P^\uparrow) \).

Note that it is sufficient to double two or three jobs to enforce this bound. An example having bad competitive ratio is as follows, and illustrated in Figure 5.3.3. First, the adversary presents \( m - 3 \) big jobs with processing time \( 1 - \frac{2}{m+1} \), which are scheduled on a machine each. Then, the adversary presents one medium job with processing time \( \frac{1}{2} - \frac{1}{m+1} - \varepsilon \), for an arbitrary small \( \varepsilon \in \mathbb{Q}_+^0 \) satisfying \( 0 < \varepsilon < \frac{1}{2(m-1)} \). Also this job is scheduled on a separate machine. Next, the adversary presents \( 2m - 2 \) small jobs with processing time \( \frac{1}{2(m+1)} \), which are scheduled evenly on the two still idle machines. Then, the adversary presents one medium job with processing time \( \frac{1}{2} - \frac{1}{m+1} \), which is scheduled on the machine processing the other medium job. Finally, the adversary presents \( 2m - 2 \) small jobs with processing time \( \frac{1}{2(m+1)} \) followed by a big job with processing time 1. The small jobs are evenly scheduled on the two machines already processing small jobs, whereas the big job is scheduled on the machine scheduling the two medium jobs. Thus, this last job defines a makespan of \( 2 - \frac{2}{m+1} - \varepsilon \). Now, we double the two medium jobs, thus making the makespan increase to \( 3 - \frac{4}{m+1} - 2\varepsilon \). The optimum offline algorithm, on the other hand, schedules all \( m \) now big jobs on different machines, and distributes the \( 4(m - 1) \) jobs with processing time \( \frac{1}{2(m+1)} \) on the remaining \( m - 1 \) machines such as to achieve a global makespan of 1. For \( m \to \infty \), the gap between the upper bound and the competitive ratio obtained for the family of examples is arbitrarily small. \( \square \)

Note that the given example can be constructed by allowing the increase of two jobs only. The result above can be extended to the
case where the processing times of the jobs are perturbed to a factor $x > 1$ of their original processing time.

**Theorem 5.19.** Consider the instances of online scheduling on $m$ identical parallel machines where perturbations may increase the processing times of many jobs to a factor $x \geq 1$, $x \in \mathbb{Q}^+$ of their original processing time. Restricted to these instances, Graham’s algorithm is $1 + x - \frac{1}{m}$-competitive. For $x \in \mathbb{N}$ and $x \leq m - 1$, the competitive ratio of Graham’s algorithm is at least $1 + x - \frac{x^2}{m-1+x} - \varepsilon$, for arbitrarily small $\varepsilon \in \mathbb{Q}^+, 0 < \varepsilon < \frac{1}{x(m-1+x)}$.

**Proof.** Consider a machine $\mu$ attaining the makespan $L_{\text{List}}^\uparrow$ as it was in the original instance. We partition the processing times of the jobs on $\mu$ as follows. Let $\alpha$ be the processing time of the last job scheduled on $\mu$, $\beta$ be the processing time of perturbed jobs excluding $\alpha$ if the last job is perturbed, and $\gamma$ be the processing time of unperturbed jobs excluding $\alpha$ if the last job is unperturbed. Due to Graham’s algorithm, $\beta + \gamma \leq L_{\text{OPT}} - \frac{\alpha}{m}$, since, on the unperturbed instance, when the last job was scheduled on $\mu$ all machines had at least this load. Thus, $\beta \leq L_{\text{OPT}} - \frac{\alpha}{m} - \gamma$. Because the processing times do only increase,
5.3. Bounded perturbations

\[ L_{\text{OPT}} \leq L_{\text{OPT}}^{\uparrow} \]. Finally, let \( \alpha^{\uparrow} \) and \( \beta^{\uparrow} \) be the perturbed counterpart of the processing times \( \alpha \) and \( \beta \). The makespan \( L(\text{List}(J, P), P^{\uparrow}) \) is

\[
L_{\text{List}}^{\uparrow} = \gamma + \beta^{\uparrow} + \alpha^{\uparrow} = \gamma + x\beta + \alpha^{\uparrow}.
\]

We distinguish two cases:

\textbf{\( \alpha \) increases:} then, \( x\alpha = \alpha^{\uparrow} \leq L_{\text{OPT}}^{\uparrow} \), since \( \alpha \) increases \( x \)-fold. Therefore, the makespan is:

\[
L_{\text{List}}^{\uparrow} = \gamma + x \cdot (\beta + \alpha) \leq xL_{\text{OPT}} + \alpha \cdot x \left( 1 - \frac{1}{m} \right) - (x - 1)\gamma
\leq xL_{\text{OPT}}^{\uparrow} + L_{\text{OPT}} \left( 1 - \frac{1}{m} \right) \leq \left( 1 + x - \frac{1}{m} \right) \cdot L_{\text{OPT}}^{\uparrow}.
\]

\textbf{\( \alpha \) does not increase:} In this case, \( \alpha \leq L_{\text{OPT}} \leq L_{\text{OPT}}^{\uparrow} \), and we have:

\[
L_{\text{List}}^{\uparrow} = \gamma + x\beta + \alpha \leq \gamma + x \cdot \left( L_{\text{OPT}} - \frac{\alpha}{m} - \gamma \right) + \alpha
\leq xL_{\text{OPT}}^{\uparrow} + \left( 1 - \frac{x}{m} \right) \alpha - (x - 1)\gamma \leq \left( 1 + x - \frac{x}{m} \right) L_{\text{OPT}}^{\uparrow}
\]

Therefore, \( L(\text{List}(J, P), P^{\uparrow}) \) \( \leq (1 + x - \frac{1}{m}) L(\text{OPT}(J, P^{\uparrow}), P^{\uparrow}) \).

An example achieving a bad competitive ratio for \( x \leq m - 1, x \in \mathbb{N} \), has the following structure, shown in Figure 5.3.4. First, the adversary presents \( m - x - 1 \) big jobs with processing time \( 1 - \frac{x}{m - 1 + x} \). The online algorithm schedules them on idle machines. Next, the adversary presents the following sequence of jobs: one medium job with processing time \( \frac{1}{x} - \frac{1}{m - 1 + x} - \varepsilon \), with \( \varepsilon \in \mathbb{Q}_+ \), \( 0 < \varepsilon < \frac{1}{x \cdot (m - 1 + x)} \), and \( (m - 1) \cdot x \) small jobs with processing time \( \frac{1}{x \cdot (m - 1 + x)} \). The same sequence is repeated \( x - 1 \) times with the difference that the medium jobs have processing time \( \frac{1}{x} - \frac{1}{m - 1 + x} \). Because of the offset \( \varepsilon \) of the first medium job, all medium jobs are scheduled on the same machine, and the small jobs are distributed evenly on \( x \) machines. At last, the adversary presents a big job with processing time 1, which is scheduled on the machine processing the medium jobs. Now, the perturbation affects the medium jobs, which increase to \( 1 - \frac{x}{m - 1 + x} \) (except for the first medium job, which is \( \varepsilon x \) smaller than that) and enforce a makespan of \( 1 + x - \frac{x^2}{m - 1 + x} - x\varepsilon \). The optimal offline algorithm, on the other hand, achieves a makespan of 1 by scheduling each big job on a different machine, and by scheduling \( x^2 \) small jobs with processing time \( \frac{1}{x \cdot (m - 1 + x)} \) on each of the \( m - 1 \) machines.
which do not yet attain the load of 1 with their big job. Note that in this example it is sufficient to perturb $x$ jobs to get this bad behavior.

In the following, we consider increases in processing times which may be different for all jobs, but where the impact on the schedule is bounded. To that end, consider Graham’s schedule $\text{List}(J, P)$. For each machine $i \in M$, we partition its assigned jobs as follows: let $\bar{j}_i$ be the last job scheduled on machine $i$, $P_i$ be the set of jobs, excluding $\bar{j}_i$, which are not perturbed, and $\bar{P}_i$ the set of jobs, excluding $\bar{j}_i$, which are perturbed.

**Theorem 5.20.** Consider the instances of online scheduling on $m$ identical parallel machines where the processing times of many jobs may increase. Assume that a machine $\mu$ attaining the makespan in a schedule obtained by Graham’s algorithm has the following properties: $\sum_{j \in \bar{P}_\mu} p_j^\uparrow = x \sum_{j \in P_\mu} p_j$ and $p_{\bar{j}_i}^\uparrow = y p_{\bar{j}_i}$. In this case, Graham’s algorithm has a competitive ratio of $1 + x - \frac{x}{ym}$, for $x \geq 1, x \in \mathbb{Q}_0^+, y \geq 1, y \in \mathbb{Q}_0^+$. 

Figure 5.3.4: The example of Theorem 5.19, here with $x = 3$. Top left, the schedule produced by Graham’s algorithm with the original instance. Bottom left, the schedule after the perturbations. Top right, the optimum offline perturbed schedule.
5.4 A Graham-like risk aversion algorithm

Proof. Consider a machine \( \mu \) attaining the makespan \( L_{\text{List}}^\uparrow \) in its unperturbed state. When Graham’s algorithm considered \( \bar{j}_\mu \), each machine had a load of at least \( \sum_{j \in \tilde{P}_\mu} p_j + \sum_{j \in P_\mu} p_j \leq L_{\text{OPT}} - \frac{p_{\bar{j}_\mu}}{m} \). Thus, \( \sum_{j \in \tilde{P}_\mu} p_j \leq L_{\text{OPT}} - \frac{p_{\bar{j}_\mu}}{m} - \sum_{j \in P_\mu} p_j \). Because the processing times increase, \( L_{\text{OPT}} \leq L_{\text{OPT}}^\uparrow \). Finally, \( y p_{\bar{j}_\mu} \leq L_{\text{OPT}}^\uparrow \), since \( \bar{j}_\mu \) must also be scheduled. The makespan \( L(\text{List}(J, P), P^\uparrow) \) can be described as follows:

\[
L_{\text{List}}^\uparrow = \sum_{j \in P_\mu} p_j + x \sum_{j \in \tilde{P}_\mu} p_j + y p_{\bar{j}_\mu}
\]

\[
\leq \sum_{j \in P_\mu} p_j + x \left( L_{\text{OPT}} - \frac{p_{\bar{j}_\mu}}{m} - \sum_{j \in P_\mu} p_j \right) + y p_{\bar{j}_\mu}
\]

\[
\leq \sum_{j \in P_\mu} p_j + x L_{\text{OPT}} - x \frac{p_{\bar{j}_\mu}}{m} - x \sum_{j \in P_\mu} p_j + y p_{\bar{j}_\mu}
\]

\[
\leq (1 - x) \sum_{j \in P_\mu} p_j + x L_{\text{OPT}} + y p_{\bar{j}_\mu} (1 - \frac{x}{ym})
\]

\[
\leq (1 + x - \frac{x}{ym}) L_{\text{OPT}}^\uparrow.
\]

Thus, \( L(\text{List}(J, P), P^\uparrow) \leq \left( 1 + x - \frac{x}{ym} \right) L(\text{OPT}(J, P^\uparrow), P^\uparrow) \). \( \square \)

5.4 A Graham-like risk aversion algorithm

In an effort to keep the impact of perturbations under control, the following strategy could prove to be effective for online scheduling. When considering the next job in the sequence, one could enumerate all possible assignments of this job to the machines. For each such assignment, we could evaluate the effect on the competitive ratio of the worst-case perturbation. Of all assignments, we could then choose the best one. Such an approach might result in an exponential running time: not only would one have to compute the optimum offline perturbed schedule in order to draw a comparison, which requires to solve an \( \mathcal{NP} \)-hard problem; more than this, such an algorithm must also determine how the worst-case perturbation looks like.
(which might be far from trivial). In an online setting an exponential-time approach is nevertheless applicable.

In this section, we consider a similar approach for bounded perturbations increasing the processing time of at most one job \( x \)-fold. The general idea of the algorithm, which we call NoRisk, is as follows. Each time a new job \( j \) of the online sequence is presented, the algorithm computes the worst-case load of each machine as follows: for a specific machine, it computes the load with the current assignment and given that \( j \) is assigned to it, and increases this load with the maximum increase in processing time resulting from perturbing any job assigned to it (i.e., the load if a worst-case perturbation in terms of additional processing time occurs). The algorithm assigns the job to the machine having least worst-case load. Thus, NoRisk is a greedy algorithm, and in each step minimizes the worst-case makespan should the sequence of jobs stop at that point, and the biggest job be perturbed. Here, the perturbation of the biggest job is seen as the perturbation which harms the schedule most. In terms of the competitive ratio this assumption is not valid, but serves as a simple greedy approach that does not require the computation of the optimal offline schedule for each assignment and perturbation.

We now describe NoRisk precisely. We number the jobs in the online sequence increasingly. Thus, the sequence is \( J = \{1, \ldots, n\} \). As specified earlier, we assume that at most one job is perturbed. The perturbation increases the job’s processing time from \( p \) to \( p^+ = x \cdot p \), for \( x > 1, x \in \mathbb{Q}_0^+ \). Let \( J_i(j), i \in M, j \in J \) be the set of jobs which have already been assigned to machine \( i \) when the job \( j \in J \) is presented but has not yet been assigned. Thus, \( J_i(1) = \emptyset, \forall i \in M \), and \( \bigcup_{i \in M} J_i(n) = J \setminus \{n\} \). We refer to the worst-case load of machine \( i \in M \) when NoRisk is considering job \( j \) as \( \ell_i(j) \). The worst-case load \( \ell_i(j) \) is defined as \( \ell_i(j) = \sum_{k \in J_i(j)} p_k + p_j + \delta_{i,j}^j \), and \( \delta_{i,j}^j = (x-1) \cdot \max \{ p_k | k \in J_i(j) \cup \{j\} \} \) is the worst-case increase in processing time on machine \( i \) if job \( j \) is assigned to that machine. Abusing notation slightly, we refer to the jobs assigned to machine \( i \in M \) when the complete sequence has been scheduled as \( J_i(n+1) \) and to the worst-case load of a machine \( i \in M \) given the assignment \( J_i(n+1) \) as \( \ell_i(n+1) \). The algorithm NoRisk is specified precisely in Algorithm 1 using the notation introduced above.

Before analyzing the algorithm, we remark that in general we would be happy with an algorithm which has a worse competitive ra-
5.4. A Graham-like risk aversion algorithm

Input: The online sequence of jobs $J = \{1, \cdots, n\}$ each with processing time $p_j, j \in J$.

Output: An online schedule assigning each job to a machine.

Initialization:

foreach $i \in M$ do $J_i(1) \leftarrow \{\}$;

Online sequence:

for $j \leftarrow 1$ to $n$ do

  for $i \leftarrow 1$ to $m$ do
    compute $\ell_i(j)$;
  end

Determine machine with least worst-case load:

$k \leftarrow \text{argmin}_{i \in M} (\ell_i(j))$;

Update assignments of jobs to machines:

foreach $i \in M$ do $J_i(j + 1) \leftarrow J_i(j)$;

$J_k(j + 1) \leftarrow J_k(j) \cup \{j\}$;

Algorithm 1: The Graham-like risk-aversion online algorithm NoRISK

The competitive ratio than Graham’s algorithm if no perturbation occurs, given that the competitive ratio is better than Graham’s if a perturbation does indeed occur. In the following, we show the competitive ratio of NoRISK if no perturbation occurs.

**Theorem 5.21.** Consider the instances of online scheduling on $m \geq 2$ parallel machines where the processing time of one job may increase to a factor $x \geq 2, x \in \mathbb{Q}_0^+$. Restricted to these instances, and if no perturbation occurs, the NoRISK scheduling algorithm has a competitive ratio of $x$.

Theorem 5.21 implies that for $x = 2$, NoRISK is almost as good as Graham’s algorithm if no perturbations occur.

**Proof.** Let $L_{\text{risk}}$ be the makespan obtained by NoRISK. We distinguish two cases. For the first, assume the job $\bar{J}_\mu$ that attains the makespan is scheduled on a machine $\mu$ that by removing $\bar{J}_\mu$ is the machine with least load. In this case, the same analysis as for Graham’s algorithm applies, since the idle times of the machines other
than $\mu$ are upper bounded by $p_{\bar{\mu}}$, $\mu$ has no idle time and the sum of the processing times are a lower bound for $m$ times the optimum value.

For the second case, we assume that the job $\bar{\mu}$ that attains the makespan $L_{\text{risk}}$ on machine $\mu$ is scheduled in such a way that by removing it from the schedule, $\mu$ is not the machine with least load. Hence, when $\mu$ starts processing $\bar{\mu}$, there is at least one machine which has been idle for some time. This situation is shown in Figure 5.4.1. Since $\bar{\mu}$ was scheduled on $\mu$, all other machines had a greater (or equal) worst-case load when NoRisk considered $\bar{\mu}$. Thus, $\ell_i(\bar{\mu}) \geq \ell_{\mu}(\bar{\mu})$, $i \in M \setminus \{\mu\}$. Now, let $\nu$ be a machine having minimum load $\ell^*$ when NoRisk considered job $\bar{\mu}$: $\ell^* = \sum_{k \in J_{\nu}(\bar{\mu})} p_k \leq \sum_{k \in J_{\nu}(\bar{\mu})} p_k, i \in M \setminus \{\nu\}$. Clearly, $\ell^* \leq L_{\text{OPT}}$, since all machines have load at least $\ell^*$ when removing $\bar{\mu}$. Let $y = L_{\text{risk}} - \ell^*$. By showing $y \leq (x - 1)L_{\text{OPT}}$, the theorem follows. Since $\bar{\mu}$ is the last job scheduled on $\mu$, we have that $\ell_{\mu}(n + 1) = \ell_{\mu}(\bar{\mu})$. Now,

$$\ell_{\mu}(\bar{\mu}) = \sum_{j \in J_{\mu}(n+1)} p_j + \delta_{\mu} = L_{\text{risk}} + \delta_{\mu} = \ell^* + y + \delta_{\mu}.$$  

Similarly, $\ell_{\nu}(\bar{\mu}) = \sum_{j \in J_{\nu}(\bar{\mu})} p_j + p_{\bar{\nu}} + \delta_{\nu} = \ell^* + p_{\bar{\nu}} + \delta_{\nu}$. Since the perturbations are bounded by a factor $x$ of the original processing time, $\delta_{j_{2,n}} \leq (x - 1) \cdot \max_{j \in J} \{p_j\} \leq (x - 1)L_{\text{OPT}}, i \in M$. Finally, by definition, $\delta_{\bar{\mu}} \geq (x - 1)p_{\bar{\mu}}$. To conclude, let us assume that

\begin{figure}
\centering
\includegraphics[width=\textwidth]{figure5_4_1.png}
\caption{The setting for the second case of the proof of Theorem 5.21. The picture reflects the situation when NoRisk was scheduling job $\bar{\mu}$. For convenience, machines are sorted according to increasing load.}
\end{figure}
5.4. A Graham-like risk aversion algorithm

$y > (x - 1) \cdot L_{\text{OPT}}$. Then,

$$\ell_\nu(j_\mu) = \ell^* + p_{j_\mu} + \delta_{j_\mu} \leq \ell^* + (x - 1) L_{\text{OPT}}$$

$$\leq \ell^* + (x - 1)p_{j_\mu} + (x - 1) L_{\text{OPT}}$$

$$< \ell^* + (x - 1)p_{j_\mu} + y \leq \ell^* + y + \delta_{j_\mu} = \ell_{\mu}(j_\mu)$$

a contradiction to having scheduled job $j_\mu$ on machine $\mu$. □

An example which comes close to the stated bound if no perturbation occurs is as follows. The online sequence is built by $m - 1$ big jobs of processing time 1, each of which is scheduled on an idle machine. These jobs are followed by a sequence of $\frac{x}{y} + 1 - x$ small jobs of processing time $y < 1$ followed by a small job of processing time $y - \varepsilon$, for an arbitrary small $\varepsilon < y$, $\varepsilon \in \mathbb{Q}^+$. All these jobs are scheduled on the same machine, since the worst-case load of a machine scheduling a big job is $x + y$, whereas the remaining machine not scheduling big jobs has a worst-case load of $ky + \frac{x - 1}{y}$ when scheduling the $k$-th small job, which is strictly smaller than $x + y$ for the whole sequence of small jobs. This schedule has a makespan of $L_{\text{risk}} = x + 2y - xy - \varepsilon$, whereas the optimum offline schedule has a makespan of $1 + \frac{1}{m}(x + 2y - 1 - xy)$. For $y \to 0$, $m \to \infty$, $\varepsilon \to 0$ and $x \ll \frac{1}{y}$ the ratio of the makespans converges to

$$\frac{x + 2y - xy}{1 + \frac{1}{m}(x + 2y - 1 - xy)} \to x$$

**Theorem 5.22.** Consider the instances of online scheduling on $m \geq 2$ identical parallel machines where the processing time of one job may increase to a factor $x \geq 2$, $x \in \mathbb{Q}_0^+$ of its original processing time. Restricted to these instances, NORISK has a competitive ratio of

$$1 + x - \frac{1}{x}.$$

**Proof.** We bound the makespan $L_{\text{risk}}$ of the perturbed instance by bounding the maximum worst-case load, which is an upper bound for any perturbed makespan. Let $\mu$ be the machine such that $\ell_{\mu}(n + 1)$ is
maximum. By Theorem 5.21, \( \sum_{j \in J_{\mu}(n+1)} p_j \leq xL_{\text{OPT}}. \)

\[
L_{\text{risk}}^\uparrow \leq \ell_{\mu}(n+1) \leq \sum_{j \in J_{\mu}(n+1)} p_j + \delta_{\mu}^{n+1} \\
\leq xL_{\text{OPT}} + (x-1) \max\{p_j : j \in J_{\mu}(n+1)\} \\
\leq xL_{\text{OPT}} + \frac{(x-1)}{x} \max\{p_j^\uparrow : j \in J_{\mu}(n+1)\} \\
\leq xL_{\text{OPT}}^\uparrow + \frac{x-1}{x} L_{\text{OPT}}^\uparrow \leq \left(1 + \frac{1}{x}\right) L_{\text{OPT}}^\uparrow.
\]

The analysis shows that \textsc{NoRisk} performs worse than Graham’s algorithm, which by comparison is slightly better than 3 competitive in this setting. The main handicap is caused by the offset which is introduced when scheduling the unperturbed instance.

An example of a bad instance and perturbation for integral \( x \in \mathbb{N} \) is as follows: the online sequence starts with \( m-1 \) jobs of processing time one. \textsc{NoRisk} schedules each of these jobs on an idle machine. Then, a sequence of \( mx^2 + m - xm - 1 \) small jobs of processing time \( \frac{1}{xm} \) is presented, followed by one small job of processing time \( \frac{1}{xm} - \varepsilon \), for \( \varepsilon < \frac{1}{xm}, \varepsilon \in \mathbb{Q}_0^+ \). All jobs are scheduled on the same machine \( \mu \). Indeed the worst-case load of \( \mu \) when the last small job is considered is \((mx^2 + m - xm)\frac{1}{xm} + (x-1)\frac{1}{xm} - \varepsilon < x + \frac{1}{x} - 1 + \frac{1}{m} - \frac{1}{xm} \leq x \) for \( m \geq 2 \) and \( x \geq 2 \), which is less than the worst-case load \( x + \frac{1}{xm} \) or \( x + \frac{1}{xm} - \varepsilon \) of the other machines. The sequence finishes with one job of processing time \( \frac{1}{x} \). This job is also scheduled on \( \mu \), since the worst-case load is \((mx^2 + m - xm)\frac{1}{xm} - \varepsilon + \frac{1}{x} + (x-1)\frac{1}{x} = x + \frac{1}{x} - 1 - \varepsilon + 1 = x + \frac{1}{x} - \varepsilon \), compared with the worst-case load \( x + \frac{1}{x} \) of the other machines. The worst-case perturbation increases the processing time of the last job in the sequence from \( \frac{1}{x} \) to \( 1 \), which results in a perturbed makespan \( L_{\text{risk}}^\uparrow = x + \frac{1}{x} - \varepsilon \). The optimum offline solution schedules the \( m \) jobs of processing time 1 (one of which is the perturbed job) on different machines, and assigns \( x^2 + 1 - x \) small jobs to each machine. In this way, the optimum offline solution has a makespan of \( 1 + \frac{x}{m} + \frac{1}{xm} - \frac{1}{m} \). For arbitrarily large \( m \to \infty \) and arbitrarily small \( \varepsilon \) the ratio of the objectives converges to \( \frac{x + \frac{1}{x}}{1 + \frac{x}{m} + \frac{1}{xm} - \frac{1}{m}} \to x + \frac{1}{x} \).

We remark the following aspects of \textsc{NoRisk}. First, for a subset of instances, \textsc{NoRisk} produces the same schedule as Graham’s
algorithm. This happens if the jobs are presented in order of increasing processing time, since in this case, for the $j$-th job in the sequence, all $\delta^i_j$, $i \in M$, are the same. Thus, the job is scheduled on the least loaded machine. Note that it is on these instances that Graham’s algorithm achieves its worst-case competitive ratio. Also, for $x = 1$, NoRisk is Graham’s algorithm. Even for $x = 2$, where NoRisk has almost the same competitive ratio as Graham’s algorithm, the previous example shows that the latter can perform better than NoRisk. Furthermore, examples can be constructed where the increase of the job with biggest processing time causes a greater makespan to NoRisk than to Graham’s algorithm: For example, on two machines and for $\varepsilon < \frac{1}{6}, \varepsilon \in \mathbb{Q}_0^+$, the sequence of jobs with processing times $(1, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, \frac{1}{6}, 1 - \varepsilon, \frac{1}{2}, \frac{1}{2}, 1, 1 + \varepsilon)$ is such a case, since the perturbation of the last (and biggest) job results in a makespan of $3.5 + \frac{2}{3}\varepsilon$ for Graham’s algorithm and in a makespan $4 - \frac{1}{3}\varepsilon$ for NoRisk.

These results show that a greedy approach on the makespan is not what we should aim for. The greatest handicap of this approach is that if no perturbations occur, the machine attaining the makespan can have (unperturbed) load nearly as high as the worst-case load of any machine.
Chapter 6

Conclusion

Throughout this thesis, we addressed both problems only from a combinatorial point of view. Even with this restricted focus, we have just scratched the surface of both problems.

Many combinatorial questions remain open for offline delay management, even with our restricted model. To mention a few, the combinatorial boundary of hard and easy instances is not yet tight. Although we suspect that the problem is solvable in polynomial time on rooted in-trees, we still lack the understanding for general trees. Furthermore, we did not address approximation algorithms in detail. From a merely practical point of view, our results point out that no polynomial-time algorithm is to be expected for the general delay management problem. Furthermore, our results show that the number of transfers of each passenger path plays a crucial role in the hardness results. Thus, the exploitation of this new insight in new solution approaches, or in classical linear programming based ones, might lead to significant improvements in the measured running times, or allow to reduce the solution space to be explored. The potential of using our combinatorial insight for solving real-life problems is yet to be explored.

The online delay management problem also allows for many extensions. Settings involving randomization were hardly analyzed in this thesis. As the delays will most certainly follow some probability distribution, the analysis of a setting reflecting probabilistic delays would be valuable in practice. In a different direction, randomization
of the online algorithm was shown to have a significant impact on ski rental and many other problems. Thus, delay management should be no exception in this regard. Finally, as a worst-case adversary does certainly not apply for real-world problems, it would be interesting to consider constrained adversaries. The restrictions on the adversary could also be derived by considering what happens in the real world. However, we point out that analyzes of the online settings for our model quickly get hard to understand. Finally, we only addressed very simple railway networks, and with a very restrictive model. A different modeling approach to delay management may allow for a clearer view of the online problem, and derive algorithms which work on more realistic railway systems.

The online parallel machine scheduling problem is just one of the possible problems that can be analyzed with respect to our measure of robustness. Naturally, many practically relevant problems are suited for our view with a problem-specific type of perturbation. This is true for both offline and online problems. Furthermore, we yet lack the understanding about what types of algorithms are robust, or how to develop algorithms which are robust with respect to a specific perturbation. Our only attempt for parallel machine scheduling showed that a greedy approach on the worst-case effect of the perturbations is not effective.

Specifically for parallel machine scheduling, a worst-case analysis of the impact of perturbations on an optimal solution would also be feasible. From a practical point of view, this analysis quantifies the impact of perturbations on the best possible solution which disregards the issue of perturbations completely. Preliminary research in this direction hints to the fact that in the worst-case, perturbations affect optimal solutions in a similar way as the solutions derived with Graham’s algorithm.

To conclude, we hope that the results in this thesis show the complexity of the analyzed optimization problems; better still, they might even stimulate the curiosity of a theoretician to see into these problems more clearly. Personally, I’ll be eagerly waiting for connections to this work.
Glossary

In this section, we briefly summarize the terminology used for delay management (in alphabetical order), which can be used as a reference in the next chapters.

**Additional passenger delay objective**  The objective function built as the sum, over all passenger paths, of each path’s arrival delay decreased by the considered path’s source delay, and weighted with the path’s weight $w$ (see page 28 for a formal definition). We aim at minimizing the additional passenger delay.

**Arrival delay**  For a given delay policy, the time difference between the time a passenger path $P$ reaches the physical destination station and the time specified by the destination node $d(P)$. For delay policies of the binary delay management problem, the arrival delay is $T$ if the passenger path is dropped, $\delta$ if it is maintained and boards a leg that waits, and zero if it is maintained and boards no leg that waits.

**Connect**  A passenger path can connect between two legs of different train services at a node $v$ if the passenger path can alight from the leg of one train service in $v$ and board a leg of the other train service at the same node.

**Consecutive legs**  Two legs are consecutive, if the terminal station of a leg coincides with the source station of the other leg.

**Delay**  The unexpected difference in time from the planned time of an event, which renders the choice on which passenger paths to maintain necessary.
Delay policy The decision of the optimization process. It specifies which legs wait and which legs depart as scheduled.

Depart as scheduled A leg departs as scheduled if it does not wait.

Drop A passenger path is said to be dropped at a node \( v \) if it misses a connection in \( v \) as a result of a delay policy. A drop happens as a consequence of an inbound leg in \( v \) waiting and of a consecutive outbound leg from \( v \) of a different train service not waiting. If a passenger path intends to board both these legs, it is dropped in \( v \).

Dynamic routing The ability of a passenger to adapt the legs it travels to the occurring delays and to the delay policy, such as to minimize her arrival delay.

Inbound leg \( e = (u, v) \) A leg \( e \) is called inbound in \( v \) if \( e = (u, v) \), which intuitively means that the train serving \( e \) arrives at node \( v \) by mean of leg \( e \).

Intermediate stop of a train service \( r \) A node is called an intermediate stop of a train service \( r \) if, for two consecutive legs \( e_1 \) and \( e_2 \) of \( r \), the node is the terminal of \( e_1 \) and the source of \( e_2 \).

Leg \( e = (u, v) \in E \) A direct connection without stops from node \( u \) to node \( v \), carried out between the two corresponding stations by one specific train service.

Node \( v \in V \) A node of the railway network represents a station at a specific time, where transfers between inbound and outbound legs can take place.

Outbound leg \( e = (u, v) \) A leg \( e \) is called outbound from node \( u \) if \( e = (u, v) \); intuitively, the train serving \( e \) departs from the station associated to node \( u \) by mean of leg \( e \).

Passenger The customer of the railway network, intending to travel from one station to a different station at a specified time, using the train services provided by the railway network.

Passenger path \( P \in \mathcal{P} \) The travel intention of a passenger, built by a sequence of consecutive legs in the network, which brings the passenger from a source node to a destination node in the network by using the specified legs.
**Passenger transfer**  The passenger’s act of alighting from a train that serves an inbound leg and boarding an outbound leg served by a different train.

**Period $T$ of the timetable**  The time interval occurring between two train services which serve exactly the same sequence of stations. In our models, we assume this time is a constant $T$, and equal for all train services.

**Serve a leg $e \in E$**  A train service $r$ serves a leg $e$ if $e \in r$. Intuitively, this happens if the train serving $r$ travels from the leg’s source station to the leg’s terminal station at the times specified by the leg.

**Serve a node $v \in V$**  A train service serves a station if it stops in node $v$. Hence, the node $v$ is the source of a leg served by the train, or the target of a leg served by the train (or both, in which case $v$ is an intermediate stop of the train service).

**Source delay $D$**  The delays given as part of the input. In our model, source delays are defined on passenger paths. In our online setting, source delays are disclosed online.

**Source delayed passenger path**  A passenger path is called source delayed if it is assigned a strictly positive delay as part of the input. Within the binary delay management problem, source delayed passenger paths have a fixed delay of $\delta$ time units. In the online setting, source delays of passenger paths are disclosed online.

**Source punctual passenger path**  A passenger path is called source punctual if it is assigned a delay zero as part of the input.

**Station**  A station is a physical place where passengers can board and alight from trains. Each node is associated to a station, a station can be associated to many different nodes.

**Total passenger delay objective**  The objective function built as the weighted sum, over all passenger paths, of each path’s arrival delay. The weight of a path $P$ is $w(P)$. The source delay of each passenger path is included in this objective function. See page 28 for a formal definition. We aim at minimizing the total passenger delay.
Train The physical mean of transport traveling in the public transportation network, built by an engine and several cars. Passengers board and alight from a train. Each train travels exactly one train service.

Train service $r \in \mathcal{R}$ The sequence of consecutive legs served by a train at a specific time. Intuitively speaking, it is the sequence of stations one trains visits, together with the time information on the departures and arrival times at the stations.

Wait The act of a leg of departing later than scheduled, either to allow a delayed passenger path to board, or to enforce consistency within a train service as a result of some previous leg in the train service waiting and the inability of train services to catch up on delays.

Weight of a passenger path The weight of a passenger path is an abstract measure of its importance. In general, it can be interpreted as the number of passengers traveling the passenger path.
Bibliography


Curriculum Vitae

Michael Joseph Gatto

Date of birth: 12.09.1977
Place of birth: Bern, Switzerland
Citizenship: Switzerland and Australia.

Education:

2003–2007 ETH Zurich, Switzerland
PhD student at the Institute of Theoretical Computer Science with Prof. Dr. Peter Widmayer.
Academic title: Dr. Sc. ETH Zurich

1997–2002 ETH Zurich, Switzerland
Student at the Department of Computer Science.
Academic title: Dipl. Inf. Ing ETH

1993–1997 Liceo Cantonale Lugano 1, Switzerland
High school, Scientific orientation.
Degree: Maturità Tipo C (high school diploma)

Practical Work

November 2001–March 2002 Comexar Engineering AG, Lachen, Switzerland.
Publications

In peer-reviewed conference proceedings:

*On the robustness of Graham’s algorithm for online scheduling* with Peter Widmayer.

*The Computational Complexity of Delay Management* with Riko Jacob, Leon Peeters and Anita Schöbel.

*Optimization of a Railway Hub-and-Spoke System: Routing and Shunting* with Riko Jacob and Marc Nunkesser.

*Online Delay Management on a Single Train Line* with Riko Jacob, Leon Peeters and Peter Widmayer.

*Railway Delay Management: Exploring its Algorithmic Complexity* with Björn Glaus, Riko Jacob, Leon Peeters and Peter Widmayer.

Other Publications

*Optimizing the Cargo Express Service of Swiss Federal Railways* with Alberto Ceselli, Marco Luebecke, Marc Nunkesser and Heiko Schilling.
Submitted for publication to Transportation Science, Focused Issue of Transportation Science on Rail Operations, 2007.

*Theory on the Tracks: A Selection of Railway Optimization Problems* with Riko Jacob, Leon Peeters, Birgitta Weber and Peter Widmayer.