Doctoral Thesis

Selected Optimization Problems in Passenger Transportation - Models, Complexity, and Algorithms

Author(s):
Böhmová, Kateřina

Publication Date:
2016

Permanent Link:
https://doi.org/10.3929/ethz-a-010781833

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.
Selected Optimization Problems in Passenger Transportation
Models, Complexity, and Algorithms

A thesis submitted to attain the degree of Doctor of Sciences of ETH Zurich

presented by
Kateřina Böhmová
Master of Science, UPC BarcelonaTech, Barcelona
Bachelor of Science, Charles University, Prague
born 17.02.1987
citizen of Czech Republic

Accepted on the recommendation of
Prof. Dr. Peter Widmayer ETH Zurich
examiner
Dr. Matúš Mihalák Maastricht University
co-examiner
Prof. Dr. Juraj Hromkovič ETH Zurich
co-examiner
Abstract

In this thesis we consider various optimization problems motivated by questions arising in passenger transportation. We propose abstract models, study the complexity of different variants of the problems, and design exact and approximation algorithms.

In Chapter 1 we study an offline interval scheduling problem where every job has exactly one associated interval on every machine. To schedule a set of jobs, exactly one of the intervals associated with each job must be selected, and the intervals selected on the same machine must not intersect. We show that deciding whether all jobs can be scheduled is NP-complete already in various simple cases, including a variant with unit-length intervals where all intervals associated with the same job have a common point. For this particular setting we propose a deterministic 501/1000-approximation algorithm. We also show that in general it is NP-hard to maximize the number of scheduled jobs even if the given instance consists of only 2 machines, and we present a deterministic 2/3-approximation algorithm for this setting.

In Chapter 2 we generalize the 2/3-approximation algorithm from Chapter 1 for more than 2 machines and for a much wider class of problems. We define a $k$-composed optimization problem $P_k$ to be composed from $k$ maximization problems $P_1, \ldots, P_k$, defined over a common ground set $S$, where the feasible solutions of each $P_i$ are subsets of $S$ that form an independence system. The goal is to find a set of disjoint feasible solutions for the $k$ problems, maximizing the size of their union. We give examples of several NP-hard optimization problems that can be seen as such a $k$-composed problem. Then, given a $\rho_i$-approximation algorithm for each $P_i$, we present a black-box approximation framework for $P_k$, which gives a $(2/3) \cdot \min_i \rho_i$-approximation algorithm for $k = 2$, and $(\frac{1}{2} + \frac{1}{2k(k-1)}) \cdot \min_i \rho_i$-approximation algorithm for $k \geq 3$. Our strategy is
simple, purely combinatorial and deterministic.

In Chapter 3 we consider an offline car-sharing assignment problem with flexible drop-offs. There are \( n \) users, each specifies one or more driving demands, consisting of a pick-up location, drop-off location, and a time interval for which the car will be used. The goal is to assign the cars, initially located at given locations, to maximize the number of users with all their demands fulfilled. If every user has exactly one demand, we show that the problem is solvable in polynomial time. If every user has two demands: one demand for transfer from location \( A \) to a location \( B \), and one demand for transfer from \( B \) to \( A \), not necessarily in this order, we show that the problem becomes APX-hard. However, if all the travel times are zero, the problem is again solvable in polynomial time.

In Chapter 4, motivated by fault-tolerance concerns in public transportation networks, we introduce sequence hypergraphs, where every hyperedge is defined as a sequence of vertices (imagine a directed path). We study the complexity of some classic algorithmic problems. In particular, we consider the problem of finding a minimum set of hyperedges that “connects” \( s \) to \( t \) (allows to travel from \( s \) to \( t \)), finding a minimum set of hyperedges whose removal “disconnects” \( t \) from \( s \), or finding two disjoint sets of hyperedges, both connecting \( s \) to \( t \). We show that many of these problems are APX-hard, even in acyclic sequence hypergraphs or with hyperedges of constant length. We then consider special settings and for some we get polynomial-time algorithms. For example, we show that if all the hyperedges are of length at most 2, these problems become polynomially solvable.
Zusammenfassung

In dieser Arbeit untersuchen wir verschiedene Optimierungsprobleme, die von Fragestellungen im Bereich der Personenbeförderung motiviert sind. Wir stellen formale Modelle vor, analysieren die Komplexität verschiedener Problemvarianten und entwerfen exakte sowie Approximationsalgorithmen.


In Kapitel 2 verallgemeinern wir den 2/3-Approximationsalgorithmus aus Kapitel 1 auf mehr als zwei Maschinen und auf eine weitere Klasse von Problemen. Dazu führen wir das Konzept eines \( k \)-zusammengesetzten Optimierungsproblems \( P^k \) ein, das aus \( k \) Maximierungsproblemen \( P_1, \ldots, P_k \) über einer gemeinsamen Grundmenge \( S \) besteht, sodass die zulässigen Lösungen für \( P_i \) Teilmengen von \( S \) sind, die ein Unabhängigkeitsystem bilden. Das Ziel besteht in der Auswahl einer Menge disjunkter Lösungen für die \( k \) Teilprobleme, deren Vereinigung maximale Kardi-
nalität hat. Wir zeigen Beispiele für verschiedene NP-schwere Optimierungsprobleme, die als $k$-zusammengesetztes Problem aufgefasst werden können. Weiterhin zeigen wir: Existiert für jedes Problem $P_i$ ein $\rho_i$-Approximationsalgorithmus, dann existiert für $P^k$ ein einfacher, kombinatorischer, determinischer Approximationsalgorithmus mit Güte $(2/3) \cdot \min_i \rho_i$, falls $k = 2$ ist, und mit Güte $(1 + \frac{1}{k(k-1)}) \cdot \min_i \rho_i$, falls $k \geq 3$ ist.


Motiviert durch Überlegungen zur Fehlertoleranz in Netzen des öffentlichen Verkehrs führen wir in Kapitel 4 Sequenz-Hypergraphen ein, in denen jede Hyperkante als Sequenz von Knoten (d.h. als gerichteter Pfad) definiert ist. Wir untersuchen die Komplexität einiger klassischer algorithmischer Probleme. Insbesondere betrachten wir die Probleme der Berechnung einer kleinstmöglichen Menge von Hyperkanten die zwei gegebene Knoten $s$ und $t$ verbindet (d.h., die es erlaubt von $s$ nach $t$ zu reisen), der Berechnung einer kleinstmöglichen Menge von Hyperkanten deren Entfernung $s$ und $t$ voneinander trennt (sodass kein $st$-Pfad mehr existiert), und der Berechnung zweier disjunkter Mengen von Hyperkanten die beide $s$ mit $t$ verbinden. Wir zeigen, dass viele dieser Probleme APX-schwer sind, selbst in azyklischen Sequenz-Hypergraphen oder wenn die Hyperkanten konstante Länge haben. Für einige Spezialfälle werden wir Polynomialzeitalgorithmen angeben. Haben zum Beispiel alle Hyperkanten eine Länge von höchstens 2, dann sind alle oben genannten Probleme in Polynomialzeit lösbar.
Acknowledgements

First and foremost, I would like to thank my supervisor Peter Widmayer, for allowing me to be a part of his group, and letting me explore the world of research in a very friendly environment. I appreciate the freedom you gave me to choose my topics, and your guidance to be able do so. Thank you for inspiration, many helpful discussions, your expertise and insights, and the words of wisdom I got whenever I needed some.

Many thanks also go to Matúš Mihalák, who is not only a co-author of my papers, but who also friendly guided me through the research and publication process, and let me gain confidence on the way. Thank you for all your advice, patience, encouragement, and valuable feedback that made me a better writer and a better researcher.

I would also like to thank Juraj Hromkovič who (together with Peter and Matúš) took the time to read and review my thesis.

At the beginning of my PhD journey, I received a lot of helpful feedback and guidance also from Yann Disser. Later, whenever there was an occasion, I was happy to work with you again.

I would like to thank Tobias Pröger for working together with me on robust routing in public transportation. It was fun to work on a more applied topic with real world data. I found the opportunity to work on the combination of that topic and the bit more theoretic topics included in this thesis just exquisite.

I also wish to thank all my co-authors for their ideas and interesting discussions that led to the results in this thesis and in our papers. You are amazing and I enjoyed very much working with you!

In our research group I felt very happy and for that I would like to thank all the people I met here. Thank you for not just being nice colleagues who
I can discuss research and other ideas with, but also good friends I can
turn to, rely on, and have fun with. Special thanks go to my closest mates,
Sandro Montanari and Tobias Pröger.

I would also like to thank Enrico Kravina and Daniel Graf, for being such
great master students. Trying to guide you was an enjoyable experience
for me and I learned a lot.

Special thanks belong to Google for the financial support in the form of the
Google Europe Fellowship in Optimization Algorithms which I received,
and to Arno Eigenwillig for being my mentor and my contact person at
Google. Also, I was supported by the EU grant FP7/2007-2013, under
grant agreement no. 288094 (project eCOMPASS).

Finally, I would like to thank all my friends and my family, in particular to
my parents Šárka and Bohouš. Thank you for all your support and love.
And to Petr Xof Škovroň, for everything.
Contents

Preface .............................................. 1
  Overview of Results .................................. 2

1 Interval Selection ..................................... 5
  1.1 Related Work .................................... 8
  1.2 Hardness of Interval Selection with Shared Cores .......... 9
    1.2.1 Interval Selection with Cores at the End ............ 11
    1.2.2 Unit Interval Selection with Cores .................... 14
  1.3 Approximating Unit Interval Selection with Cores beyond 1/2 19
    1.3.1 Standard techniques fail ......................... 20
    1.3.2 The approximation algorithm ...................... 21
    1.3.3 Analysis .................................... 27
  1.4 Hardness of Interval Selection with Small Number of Machines 33
    1.4.1 Unit Interval Selection with Three Machines ........ 34
    1.4.2 Unit Interval Selection with Two Machines .......... 36
  1.5 Approximating Interval Selection with Two Machines ........ 39

2 Composed Problems .................................... 43
  2.1 Approximating a Maximum Red-Blue Split Subgraph .......... 45
  2.2 Composed optimization problems .......................... 47
    2.2.1 Examples of composed optimization problems ........ 48
  2.3 Approximating composed problems .......................... 50
    2.3.1 Approximating 2-composed maximization problems ...... 50
    2.3.2 Approximating $k$-composed optimization problems ... 53
    2.3.3 Upper bounds on the approximation ratio of the algorithm .............. 58
  2.4 Discussion and Related Work ............................ 62
In this thesis, we consider various algorithmic problems that originated as elementary questions, when we were wondering about different aspects of passenger transportation. In the following, I give you examples of the basic questions that absorbed our attention.

When traveling from a place $A$ to a place $B$ using a public transportation network in a city, sometimes there is an outage of a transportation line (e.g., a tram line) and one has to take a detour, using other lines. Motivated by an interest in the network’s resilience to failures, we wondered: How many lines must fail so that there is no longer a way to reach $B$ from $A$? Are there two ways to get from $A$ to $B$ that do not use the same transportation lines? How many lines do I need to use to travel from $A$ to $B$?

As a user of a car-sharing service, one wants to rent one of the available cars for a certain time period. Often, there are several locations in the neighborhood and it may be acceptable for the user to rent the car from any of these locations. Say, every user specifies for each location some time period when she would need the car if taken from that location, but at the end she is satisfied if she gets assigned a car from any location. Then, questions like the following arise: Given the demands, is it possible to assign cars to users to satisfy them all? If not, how to satisfy as many of them as possible?

Sometimes, the users of a car-sharing service would like to drive with the car from one location to another, to be flexible on where to return the car. And if that is an option, they may be interested in using the car for their daily commute – travel with the car from one location to another in the morning, and back in the afternoon. Then again, we wondered: If everybody has this type of request, how should we assign the cars to the users, to satisfy as many of them as possible?
In this work, we look at the arising questions from an algorithmic perspective: we propose abstract models, study the computational complexity of different variants of the problems, and provide efficient algorithms that give optimal or close to optimal solutions.

Overview of Results

The thesis is organized into 4 chapters. In each chapter we consider and discuss a different optimization problem, generally motivated by the above outlined questions arising in passenger transportation. Here we give an overview of the problems we study and the results we derive. At the beginning of each chapter we describe both the setting and the results in more detail. The results have been developed in a joint work with a number of people and have been published or submitted for publication; we give more detailed information in each respective chapter.

In Chapter 1 we consider the above mentioned optimization problem arising in a car-sharing service where every user specifies a time period for each location but accepts a car from any location, with the objective of maximizing the number of satisfied users. We formulate the problem as an offline interval scheduling problem where every job (user) has exactly one associated time interval on every machine (car). To schedule a set of jobs, exactly one of the intervals associated with each job must be selected, and the intervals selected on the same machine must not intersect. The goal is to find a schedule maximizing the number of scheduled jobs.

We show this problem to be NP-hard already in various simple cases. In particular, we consider the case when all the intervals associated with the same job end at the same point in time (also known as just-in-time jobs), and show that deciding whether all jobs can be scheduled is NP-complete. This solves an open problem posed by Sung and Vlach [62, 65]. Furthermore, we show that deciding whether all jobs can be scheduled is NP-complete also for the variant with unit-length intervals where all intervals associated with the same job have a common point, and for the variant with unit-length intervals and three machines. Then we consider the setting with two machines and unit-length intervals and prove that maximizing the number of scheduled jobs is NP-hard.

For some variants of the problem we design deterministic approximation algorithms. In particular, we propose a $501/1000$-approximation algorithm for the setting with unit-length intervals and all intervals of each job
 having a point in common. For the variant with two machines (and intervals of arbitrary lengths) we present a 2/3-approximation algorithm. Both algorithms improve upon the previously known deterministic 1/2-approximation algorithm.

In Chapter 2 we generalize the idea of the 2/3-approximation algorithm from Chapter 1 for a much wider class of problems.

We consider the general question of optimizing several maximization problems $P_1, \ldots, P_k$ at once, defined over the same ground set $S$ for which feasible solutions of every problem $P_i$ are subsets of $S$ that form an independence system. This joint optimization problem asks to find pairwise disjoint feasible solutions $F_1, F_2, \ldots, F_k$ for the respective problems so as to maximize $\sum_{e \in F_1 \cup F_2 \cup \ldots \cup F_k} w(e)$, where $w(e)$ is the weight of element $e$.

We show that many NP-hard optimization problems can be seen as such a $k$-composed optimization problem, including the maximum red-blue split subgraph, the interval selection problem, and the maximum edge-$k$-coloring.

We then present a black-box approximation framework for approximating a $k$-composed problem $P_k$ that reuses existing exact or approximation algorithms for the individual maximization problems $P_1, \ldots, P_k$. In particular, assuming a $\rho_i$-approximation algorithm for $P_i$, we obtain for $P_k$ a $(2/3) \cdot \min_i \rho_i$-approximation algorithm for $k = 2$, and $(\frac{1}{2} + \frac{1}{2k(k-1)}) \cdot \min_i \rho_i$-approximation algorithm for $k \geq 3$. The running-time overhead of the framework is a multiplicative $k!$-factor. By applying this framework to the example problems mentioned above, we provide new and/or improved approximation algorithms. Similarly to the idea behind the 2/3-approximation algorithm from Chapter 1, the strategy presented in Chapter 2 is extremely simple, purely combinatorial and deterministic.

In Chapter 3 we again consider optimization problems motivated by assignment problems in car-sharing, this time we want to allow the users to transfer with the rented car from one location to another and leave the car there. Each of $n$ users presents their driving demands, and the goal is to assign the cars, initially located at given locations, to maximize the number of satisfied users. Each driving demand of a user specifies the pick-up location and the drop-off location, as well as the time interval in which the car will be used. If a user requests several driving demands, then she is satisfied only if all her demands are fulfilled (partial rentals are not possible).

We show that minimizing the number of vehicles that are needed to fulfill
all demands is solvable in polynomial time. If every user has exactly one demand, we show that for any number of cars and locations, maximizing the number of satisfied users is also solvable in polynomial time. We then study the problem with two locations $A$ and $B$, and where every user has two demands: one demand for transfer from $A$ to $B$, and one demand for transfer from $B$ to $A$, not necessarily in this order. We show that maximizing the number of satisfied users is NP-hard, and even APX-hard, even if all the transfers take exactly the same (non-zero) time and there is only one car. On the other hand, if all the transfers are instantaneous, we prove that the problem is again solvable in polynomial time.

In Chapter 4 we consider optimization problems arising in public transportation networks, motivated by the interest in the network’s resilience to line failures. To conveniently model the network for our purposes, we introduce sequence hypergraphs by extending the concept of a directed edge (from simple directed graphs) to hypergraphs. Specifically, every hyperedge of a sequence hypergraph is defined as a non-repeating sequence of vertices (imagine it as a directed path), and serves to represent a transportation line (in one direction).

We study the complexity of some classic algorithmic problems, arising (not only) in transportation, in the setting of sequence hypergraphs. In particular, we consider the problem of finding a shortest $st$-hyperpath: a minimum set of hyperedges that allows to travel from $s$ to $t$; finding a minimum $st$-hypercut: a minimum set of hyperedges whose removal “disconnects” $t$ from $s$; or finding a maximum $st$-hyperflow: a maximum number of hyperedge-disjoint $st$-hyperpaths.

We show that many of these problems are APX-hard, even in acyclic sequence hypergraphs or with hyperedges of constant length. However, if all the hyperedges are of length at most 2, we show that these problems become polynomially solvable. We also study the special setting in which for every hyperedge there also is a hyperedge with the same sequence, but in the reverse order (which often happens in public transportation networks). We show that the shortest $st$-hyperpath problem becomes polynomially solvable, but both the maximum $st$-hyperflow problem and the minimum $st$-hypercut problem remain NP-hard also in this setting, and we show that there is a $1/2$-approximation algorithm for the minimum $st$-hypercut problem. Finally, we briefly discuss other algorithmic problems (e.g., finding a minimum spanning tree, or connected components).
Chapter 1

Interval Selection

We consider a fixed-interval scheduling problem with \( m \) machines and \( n \) jobs. A job consists of \( m \) open intervals – each associated with exactly one machine. In other words, each job has exactly one interval on each machine. To schedule a job, exactly one of its intervals must be selected. To schedule several jobs, no two selected intervals on the same machine may intersect. The goal is to schedule the maximum number of jobs. We will refer to this problem as IntervalSelection.

The presented problem (much like general interval scheduling problems) is motivated by real-world applications, see, e.g., [5, 19, 30]. Our motivation comes from the area of car-sharing where several users (jobs) wish to reserve a car (machine) for a certain amount of time (interval), sufficiently large to drive to an appointment location (specific to each user) and back. The distance of the parking place of each car to the destination may vary, and this results, for each user, in various time intervals for the cars.

In this chapter we study the complexity and approximation of different variants of IntervalSelection. We show that this problem is NP-hard already in very restricted settings, and we give deterministic approximation algorithms for some of the variants.

In the special case of a single machine, our problem becomes the classic interval scheduling problem which is solvable in \( O(n \log n) \) time by the following greedy algorithm: Scan iteratively the right endpoints of the intervals from left to right, and in each iteration select the considered interval, if and only if it does not intersect any of the previously selected
Chapter 1. Interval Selection

intervals. We will refer to this algorithm as the single-machine greedy.

For the case of two machines, it can be decided in polynomial time whether all jobs can be scheduled (by a reduction to 2-SAT). In contrast to this, in the present chapter we show that the same question is NP-complete for the case of three machines (Section 1.4.1). Moreover, we show that the problem of maximizing the number of scheduled jobs is NP-hard already for two machines (Section 1.4.2). Both results hold even if all the intervals have unit length.

We also consider variants of INTERVAL SELECTION where all intervals of the same job, when seen on the real line, have a non-empty intersection (e.g., this would be the time around the user’s appointment in the mentioned car-sharing application). We call such a non-empty intersection a core of a job. We refer to INTERVAL SELECTION where each job has a core as INTERVAL SELECTION with cores. A special case of such a variant is when all intervals of a job have the same end-point (so called just-in-time jobs [65]). We show that, in this setting, the problem of deciding whether all jobs can be scheduled is NP-complete (Section 1.2.1). This solves an open problem posed by Sung and Vlach [62, 65]. If the cores do not have to be at the right-end of the intervals, we show that deciding whether all jobs can be scheduled is NP-complete already when all intervals have unit length (Section 1.2.2).

Our problem can be seen as a special case of another problem, studied under the name job interval selection problem, denoted as JISP_k. There, each job has k associated intervals on the real line, not divided into machines, and the goal is again to schedule the maximum number of jobs, by picking one interval for each, without intersections. To see INTERVAL SELECTION with m machines as a special case of JISP_m, consider the machines of an instance of INTERVAL SELECTION in any order, and just concatenate the intervals for the machines along the real line, thus creating an instance of JISP_m. JISP_k is APX-hard for any k \ge 2, and only a deterministic 1/2-approximation algorithm is known (in fact, a simple greedy algorithm) [64], and a randomized \approx e^{-1} -approximation algorithm [19] that gives a 3/4-approximation for JISP_2. Both these algorithms can be directly applied to our problem. In particular, the deterministic 1/2-approximation algorithm can be described for our problem as a straightforward generalization of the single-machine greedy: Consider the machines one by one in an arbitrary order, run the single-machine greedy on the intervals of the currently considered machine, add all the selected intervals to the solution and remove the jobs that correspond to them from all the subsequent machines. We will refer to this algorithms as the multi-machine greedy.
Table 1.1: Summary of the complexity of INTERVAL SELECTION problems with $n$ jobs, and $m$ machines. The cells in gray indicate our contribution.

<table>
<thead>
<tr>
<th></th>
<th>Schedule all jobs</th>
<th>Max # jobs</th>
<th>Max $\sum$ weights</th>
</tr>
</thead>
<tbody>
<tr>
<td>single machine</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
</tr>
<tr>
<td>identical intervals per job</td>
<td>$O(n \log n)$</td>
<td>$O(n \log n)$</td>
<td>$O(n^2 \log n)$</td>
</tr>
<tr>
<td>with cores, any $m$</td>
<td><strong>NP-complete</strong> † §</td>
<td><strong>NP-hard</strong> † §</td>
<td><strong>NP-hard</strong> † §</td>
</tr>
<tr>
<td></td>
<td>$O(mn^{m+1})$</td>
<td>$O(mn^{m+1})$</td>
<td>$O(mn^{m+1})$</td>
</tr>
<tr>
<td>no core required</td>
<td><strong>NP-complete</strong> †</td>
<td><strong>NP-hard</strong> †</td>
<td><strong>NP-hard</strong> †</td>
</tr>
<tr>
<td>2 machines</td>
<td>$O(n^2)$</td>
<td><strong>NP-hard</strong> †</td>
<td><strong>NP-hard</strong> †</td>
</tr>
<tr>
<td>$\geq 3$ machines</td>
<td><strong>NP-complete</strong> †</td>
<td><strong>NP-hard</strong> †</td>
<td><strong>NP-hard</strong> †</td>
</tr>
<tr>
<td>JISP$_k$ (single machine)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2 intervals per job</td>
<td>$O(n^2)$</td>
<td><strong>NP-hard</strong> †</td>
<td><strong>NP-hard</strong> †</td>
</tr>
<tr>
<td>$\geq 3$ intervals per job</td>
<td><strong>NP-complete</strong> †</td>
<td><strong>NP-hard</strong> †</td>
<td><strong>NP-hard</strong> †</td>
</tr>
</tbody>
</table>

$§$ even if
- all cores at the end, or
- all cores in the middle

$†$ even if all intervals have unit length

Table 1.1 provides an overview of the known (white background) and new (grey background) complexity results for INTERVAL SELECTION and related problems. The columns distinguish three basic computational goals: scheduling all jobs, the maximum number of jobs, or jobs of maximum weight. Each row block, from top to bottom, is a generalization of the problem in the previous row block, starting with INTERVAL SELECTION on a single machine, and ending with JISP$_k$. As can be seen from the table, the (general) INTERVAL SELECTION, denoted as “no core required” in the table, is closely related to well-known and studied problems: it offers a natural generalization of the setting “with cores” [62, 65], and it is an interesting special case of JISP$_k$ [19, 30, 64]. Previous work left a gap in the understanding of the complexity of the problems (the grey areas in the table), which we address and completely close in this chapter. To achieve tight hardness results for the boundary cases of 2 and 3 machines (for the decision variant), or 1 and 2 machines (for the maximization objective), we devise gadgets that we plug together using known results on a specific graph coloring problem (solvable in polynomial time), which might be of independent interest. Notably, where meaningful, the hardness results hold even if all intervals are of unit length.

On the positive side, as we already mentioned, there is a deterministic 1/2-approximation algorithm for INTERVAL SELECTION. In this chapter, we present deterministic approximation algorithms that beat this barrier of 1/2 for some variants of INTERVAL SELECTION. In particular, in Section 1.3 we
describe a deterministic algorithm for INTERVALSELECTION with cores that achieves an approximation ratio 501/1000, independently of the number of machines \( m \) in the given instance of the problem. In Section 1.5 we present a simple deterministic 2/3-approximation algorithm for INTERVALSELECTION with two machines. Later on, in Chapter 2, we will see how the simple and elegant idea behind this algorithm can be further generalized for more than 2 machines and beyond INTERVALSELECTION.

The results in this chapter are a joint work with my co-authors. In particular, the 501/1000-approximation algorithm given in Section 1.3 is a joint work with Enrico Kravina and Matúš Mihalák. Parts of this section were written by Enrico Kravina and appeared in his Master thesis [52], which I co-supervised. It was published in ISCO 2016 [13]. The remaining results in this chapter, namely the negative results as well as the 2/3-approximation algorithm, are a joint work with Yann Disser, Matúš Mihalák, and Peter Widmayer. They were published in WADS 2013 [11].

1.1 Related Work

The general interest in interval scheduling problems dates back to the 1950s. The classical variant, in which each job has associated an interval and can be scheduled on any of the machines (i.e., in our setting, each job has exactly the same interval on every machine) and the goal is to decide whether all the jobs can be scheduled, is polynomially solvable [1]. The maximization version is polynomially solvable as well, even if the jobs are weighted [15]. However, Arkin and Silverberg [1] showed that if each job can only be scheduled on a subset of the \( m \) machines, the problem becomes NP-hard (even in the unweighted case). They also gave a \( O(n^{m+1}) \)-time algorithm (i.e., polynomial for a constant \( m \)).

The special case of our problem with just-in-time jobs (i.e., where all intervals of a job have the same right end point) has been studied by Sung and Vlach [65]. They showed that the weighted version is NP-hard and presented a dynamic programming algorithm that solves the problem in time \( O(m \cdot n^{m+1}) \). Settling the complexity of the problem with unit-weight jobs was posed as an open problem [65]; this open problem has also been stated in a recent survey on just-in-time job scheduling [62]. After we solved this problem in [11] (Section 1.2.1), recently Passchyn et al. [58] studied a more specific variant of the setting with just-in-time jobs. In particular, they considered a variant where all the intervals on one machine
are of the same length and showed that also in this case it is NP-complete to decide whether all jobs can be scheduled. This strengthens our previously given result.

As outlined beforehand, our problem is a special class of JISP\(_k\) (job interval scheduling problem on a single machine with \(k\) intervals per job). Nakajima and Hakimi [55] showed that the decision version of JISP\(_3\) is NP-complete. Keil [46] showed that this is the case even if the intervals have the same length, while the general decision version of JISP\(_2\) can be solved in polynomial time. The maximization version has been studied as outlined earlier by Spieksma [64] and Chuzhoy [19]. Erlebach and Spieksma [30] consider the weighted JISP\(_k\) with more than one machine (every job has the same set of \(k\) intervals on every machine) and they study myopic (single-pass) greedy algorithms.

JISP\(_k\) is, in some sense, a discrete variant of the throughput-maximization problem (also known as the time-constrained scheduling problem, or the real-time scheduling problem), in which each job has a length, a release time, and a deadline, and a job is associated with the (infinite) set of intervals of given length lying between the job’s release time and the deadline. Bar-Noy et al. [5] study this problem and give the currently best approximation algorithms for most of the existing variants of the problem.

There are many other, for the scope of the chapter less relevant variants of scheduling where intervals “come into play”. We refer to the surveys by Kolen et al. [48] and by Kovalyov et al. [51] for more information on the topic. We also stress that online variants of the presented problems have been studied as well, see e.g., the recent paper of Sgall [61] on online throughput maximization.

1.2 Hardness of Interval Selection with Shared Cores

In this and the following section we consider a special case of INTERVAL-SELECTION where we require that for every job, there is a point in time such that every interval of this job intersects with this time-point (on its corresponding machine). Recall, we refer to this time-point as the core of the job.

In this section we study the complexity of INTERVAL-SELECTION with cores. In particular, we study two variants. First, we consider the case when
every job has a core at the end, i.e., all intervals of a job end at the same point in time. We show that deciding whether there is a complete schedule for this variant is NP-complete. By this we resolve an open problem posed by Sung and Vlach [62, 65]. Afterwards, we consider the case where every job has a core at an arbitrary position and show that this variant is NP-complete even if all intervals have unit length. We note that both variants are solvable in time $O(m \cdot n^{m+1})$, and thus in polynomial time if $m$ is constant [65].

We also note that using the exact same hardness constructions as we will describe in this section, one can even prove both the problems to be APX-hard, as Kravina [52] showed in his Master thesis. Nevertheless, for simplicity of exposition, we present our original NP-hardness proofs.

Let us first give some definitions that will be used also in the later sections of this chapter. By an interval we understand a particular time interval corresponding to a job on a machine. However, in the following, we will also use time intervals not associated with a job or a machine. To avoid confusion, let us formalize the terminology with the following definition.

**Definition 1.1 (Window, Slot, Interval).** When we consider a time interval independently of machines and jobs, we call it a *window*. When considering a time interval with respect to a single machine, but independently of the jobs, we call it a *slot*. Finally, to a time interval associated with both a job and a machine we simply refer as an *interval*.

**Definition 1.2 (Blocking).** We say that an interval $i$ blocks a slot $s$ if $i$ intersects $s$ and both are associated with the same machine. We say that a set of intervals $I$ blocks window $w$ on a set of machines $\mathcal{M}$ if for each machine $M$ in $\mathcal{M}$ there is an interval in $I$ that blocks the slot corresponding to $w$ on $M$. We say that a set of intervals $I$ completely blocks a window $w$ if each slot that intersects the window $w$ is blocked by some interval in $I$.

**Definition 1.3 (Complete Schedule).** We call a schedule in which all jobs are scheduled a *complete schedule*.

Our hardness results in this chapter are shown by a reduction from variants of the NP-complete *satisfiability problem* (SAT). SAT is the problem of finding, for a given a set of $r$ clauses $C = \{c_1, c_2, \ldots, c_r\}$ over a set of Boolean variables $X = \{x_1, x_2, \ldots, x_s\}$, a truth assignment such that every clause is satisfied, i.e., at least one literal in every clause evaluates to TRUE (see, e.g., [39] for an exact definition of the problem). SAT is NP-complete, even if every clause is restricted to have at most three literals (denoted as 3-SAT) [39], and even, if each clause contains at most three literals and each
variable appears in the formula at most three times, once as a negative literal and at most twice as a positive literal (denoted as \((\leq 3,3)\)-Sat) [39]. The problem of finding a truth assignment that maximizes the number of satisfied clauses is NP-hard, even if each clause contains two literals and each variable appears at most three times in the formula (denoted as \((2,3)\)-MaxSat) [59].

1.2.1 Interval Selection with Cores at the End

We show that given an instance of INTERVAL SELECTION with cores at the end it is NP-complete to decide whether all jobs can be scheduled.

To simplify the hardness construction, let us first define the following gadget. We will reuse this gadget also in hardness constructions in later sections of this chapter.

**Definition 1.4** (Blocking Gadget for Window \(w\)). For a given window \(w\), the blocking gadget for \(w\) consists of \(m\) jobs, each having \(w\) as their interval on every machine.

The purpose of this gadget is to completely block a certain window \(w\), i.e., to make sure that in any complete schedule no interval that intersects \(w\) is ever scheduled, with the exception of the intervals of the jobs that constitute the gadget itself. We visually depict a blocking gadget as in Figure 1.1.

**Lemma 1.5.** In any complete schedule for an instance of INTERVAL SELECTION that contains the blocking gadget \(B\) for window \(w\), no selected interval outside \(B\) intersects \(w\).

**Proof.** In any complete schedule all the jobs have to be scheduled, including
all the jobs of the blocking gadget. Each of these jobs can only be scheduled onto $w$ and there are as many jobs as machines, so there is no other way than to schedule exactly one of the jobs onto the window $w$ on each machine. Thus, the selected intervals of these jobs completely block window $w$ and no other interval intersecting $w$ can be selected in the schedule.

With the blocking gadget defined, let us now proceed to prove the following theorem.

**Theorem 1.6.** The problem of deciding whether there exists a complete schedule for INTERVALSELECTION with cores at the end is NP-complete.

**Proof.** The problem is in NP, since the completeness of a given schedule can be checked in linear time. To show the hardness, we present a reduction from 3-SAT.

**Construction.** Let us consider an arbitrary instance $\Phi$ of 3-SAT given by a set of clauses $C = \{c_1, c_2, \ldots, c_r\}$ over a set of Boolean variables $X = \{x_1, x_2, \ldots, x_s\}$. We construct the following instance $S$ of the INTERVALSELECTION problem (cf. Figure 1.2 along with the construction). We use two machines for each variable $x_i$, denoted by $M_{x_i,+}$ and $M_{x_i,-}$. The machine $M_{x_i,+}$ corresponds to the positive literal of $x_i$, whereas $M_{x_i,-}$ corresponds to the negative literal of $x_i$. On the machines we consider a window of $r + 1$ time units and we denote the unit windows constituting it by $w_0, w_1, w_2, \ldots, w_r$. We place a blocking gadget over all machines on the window $w_0$. Next, for each variable $x_i$ we add a job $\alpha_{x_i}$ with two possible ways of scheduling it (in any complete schedule). This mimics a truth assignment to the variable $x_i$. We call these jobs the **variable jobs**. We place the intervals of a variable job $\alpha_{x_i}$ as follows. On $M_{x_i,+}$ and $M_{x_i,-}$ we place an interval such that it covers $w_1, w_2, \ldots, w_r$, and on every other machine we place an interval such that it covers $w_0, w_1, w_2, \ldots, w_r$. Note that the blocking gadget ensures that in any complete schedule each job $\alpha_{x_i}$ is scheduled on one of the machines $M_{x_i,+}$, $M_{x_i,-}$, and no other job is scheduled on that machine on any window $w_1, w_2, \ldots, w_r$. By scheduling $\alpha_{x_i}$, one of the two literals of $x_i$ is selected and thus set to FALSE, implicitly setting a truth assignment for variable $x_i$. Lastly, we add $r$ jobs linked to the clauses so that the actual scheduling of these jobs is related to the way how the clauses of $\Phi$ are satisfied. For each clause $c_j$ we have one **clause job** denoted by $\beta_{c_j}$. We place the intervals for the job $\beta_{c_j}$ on window $w_j$ on those machines that correspond to literals that appear in the clause $c_j$, and on the windows $w_0, w_1, \ldots, w_j$ on the other machines. In other words,
Figure 1.2: Example of the construction of INTERVAL SELECTION with cores at the end for an instance $\Phi$ of the 3-SAT problem (each figure shows the intervals on a single machine), where $\Phi = (x_1 \lor x_2 \lor \overline{x}_3) \land (x_1 \lor x_3 \lor \overline{x}_4) \land (\overline{x}_2 \lor \overline{x}_3 \lor x_4) \land (x_1 \lor \overline{x}_3 \lor x_4)$.

in any complete schedule, a job $\beta_{c_j}$ can only be scheduled on a machine that corresponds to a literal that appears in clause $c_j$, since on all other machines the intervals for $\beta_{c_j}$ intersect the blocking gadget. Moreover, if the same literal appears in clauses $c_j$ and $c_{j'}$, $j \neq j'$, then the intervals for jobs $\beta_{c_j}$ and $\beta_{c_{j'}}$ do not intersect on the machine that corresponds to this literal.

Note that the constructed instance of INTERVAL SELECTION has the property that all the intervals corresponding to one job have at their end a unit window in common, thus, it is an instance of INTERVAL SELECTION with cores at the end. Obviously, the above construction can be done in polynomial
time.

Correctness. We now show that $\Phi$ is satisfiable if and only if there exists a complete schedule for $S$. First, given a complete schedule $S$ for $S$, we construct a satisfying truth assignment $A$ for $\Phi$ as follows. The blocking gadget ensures that no selected interval for a variable or clause job intersects window $w_0$. Since $S$ is a complete schedule, all the variable jobs are scheduled. Due to the blocking gadget, each variable job $\alpha_{x_i}$ can only be scheduled on $M_{x_i,+}$ or $M_{x_i,-}$; and we set the variable $x_i$ in assignment $A$ to FALSE or TRUE, respectively. We argue that all clauses are satisfied by $A$. Let $c_j$ be any clause of $\Phi$. As $S$ is a complete schedule, the clause job $\beta_{c_j}$ that corresponds to $c_j$ is scheduled on $M_{x_i,+}$ or $M_{x_i,-}$ for some variable $x_i$ appearing in $c_j$. Assume $x_i$ appears as a positive literal in $c_j$. Then $\beta_{c_j}$ must be scheduled on $M_{x_i,+}$. Also, the variable job $\alpha_{x_i}$ has to be scheduled on $M_{x_i,-}$, since the intervals for jobs $\alpha_{x_i}$ and $\beta_{c_j}$ on $M_{x_i,+}$ overlap. But this means that $A$ sets $x_i$ to TRUE and hence satisfies $c_j$. Analogously, $A$ satisfies $c_j$ also if $x_i$ appears as a negative literal in $c_j$. Conversely, we can construct a complete schedule from a truth assignment $A$ that satisfies $\Phi$ as follows. We schedule all the jobs that form the blocking gadget on $w_0$ in any order. Then, we schedule all variable jobs according to $A$: If a variable $x_i$ is set to TRUE, we schedule $\alpha_{x_i}$ on machine $M_{x_i,-}$, otherwise on machine $M_{x_i,+}$. Lastly, as all the clauses are satisfied by assignment $A$, each clause $c_j$ is satisfied by some variable $x_i$. Either $x_i$ appears in $c_j$ as a positive literal, or as a negative literal. In the former case, $x_i$ is set to TRUE in $A$ and machine $M_{x_i,+}$ is not occupied by $\alpha_{x_i}$, so the clause job $\beta_{c_j}$ can be scheduled on machine $M_{x_i,+}$. In the latter case, the clause job $\beta_{c_j}$ can be scheduled on $M_{x_i,-}$. We can construct a complete schedule since no two scheduled clause jobs overlap.

The presented hardness implies the hardness of other variants of INTERVALSELECTION, such as that of cores at arbitrary positions, or with no required core at all. Similarly, the presented hardness implies the hardness of the maximization versions of these variants.

1.2.2 Unit Interval Selection with Cores

A very special case of INTERVALSELECTION is the setting where all intervals are of the same length. The previous hardness result does not imply the hardness of this case. Here we show that deciding whether there is a
Figure 1.3: Visual representation of decision gadgets on three machines $M_0$, $M_1$, and $M_2$. Each of the decision gadgets has two positive slots on machines in $Q_+$ and one negative slot on the machine in $Q_-$. The crucial intervals constituting the gadget are depicted by the shaded boxes (always one interval spans the respective box). The associated jobs of the intervals are indicated on the sides. The remaining intervals of the jobs are blocked by a blocking gadget, and thus never selected. These intervals and the blocking gadget are for simplicity not displayed. The two different shades in the boxes depict the only two possibilities how to select the intervals in the decision gadget.

complete schedule for INTERVAL SELECTION with cores is NP-complete even in this special case.

To simplify the hardness construction, besides using the blocking gadget defined in Section 1.2.1, we also define and use the following gadget (see Figure 1.3 for the visual representation). Again, we will reuse this gadget also in hardness constructions in the later sections of this chapter.

**Definition 1.7** (Decision Gadget). Given a window $w$ of unit length and a tuple $(Q_-, Q_+)$ of disjoint subsets of machines, we call the window $w$ on the machines in $Q_+$ the positive slots and $w$ on $Q_-$ the negative slots of the gadget. We construct the decision gadget as follows. We denote by $Q$ the union of $Q_-, Q_+$, by $k$ the size of $Q$, and by $M_0, M_1, \ldots, M_{k-1}$ the machines in $Q$. We use $k$ jobs $j_0, j_1, \ldots, j_{k-1}$, one job per machine in $Q$. The intervals for all these jobs have unit length $|w|$. We assume there is a blocking gadget $B$ such that all intervals of the decision gadget except for intervals of $j_i$ on $M_i, M_{i-1}$ intersect $B$ (we write $M_{i-1}$ instead of $M_{i-1 \mod k}$ for simplicity). The exact placement of $B$ is specified later, during the hardness construction. The exact placement of $j_i$ and $j_{i+1}$ on $M_i$ depends on whether the window $w$ is supposed to be a positive or a negative slot on $M_i$. In particular, if $M_i$ is in $Q_-$ ($w$ is a negative slot on $M_i$), the interval for $j_i$ is placed directly to the right of $w$ and the interval for $j_{i+1}$ is placed...
so that its left end is at the center of \( w \). Otherwise, if \( M_i \) is in \( Q_+ \), the left end of the interval for \( j_i \) is at the center of \( w \) and the interval for \( j_{i+1} \) is directly to the right of \( w \).

The purpose of the decision gadget is to mimic a truth assignment to a variable in a boolean formula of 3-SAT. This is done by blocking a certain window either on one set of machines or on another disjoint set. With our gadget we want to achieve that in any complete schedule either all the positive slots of the gadget are free and all the negative slots are blocked by the schedule, or vice versa. Let us refer to the former situation as the positive decision of the gadget and to the latter as the negative decision. Intuitively, we achieve this effect by using jobs with intervals placed so that we have exactly two ways how to schedule all jobs. To ensure that there is no other way to schedule the jobs of the gadget, we may need to block some intervals of these jobs. For this purpose we use a blocking gadget.

Note that the intervals constituting the gadget together with the slots occupy a window of length 2 (excluding the intervals that are blocked by the blocking gadget).

**Lemma 1.8.** In any complete schedule for an instance of Interval Selection that contains the decision gadget \( D \) for window \( w \) and subsets \( Q_-, Q_+ \) of machines, either \( D \) blocks \( w \) on all machines in \( Q_- \) and leaves it free on all machines in \( Q_+ \), or vice versa.

**Proof.** We observe that the interval for job \( j_i \) intersects with the interval for job \( j_{i+1} \) on machine \( M_i \) and with the interval for job \( j_{i-1} \) on machine \( M_{i-1} \). Furthermore, because of the blocking gadget, \( j_i \) cannot be scheduled on any other than these two machines. First, let us assume that in a complete schedule \( S \) job \( j_i \) is scheduled on machine \( M_i \). The schedule \( S \) needs to schedule job \( j_{i+1} \), which can only be scheduled on machine \( M_{i+1} \). A similar situation occurs for job \( j_{i+2} \). In fact, since \( S \) is a complete schedule, the initial decision is propagated over all the jobs of the gadget. Conversely, if we assume that a complete schedule \( S \) schedules a job \( j_i \) on machine \( M_{i-1} \), then each job \( j_j \) must be scheduled on machine \( M_{j'-1} \). Finally, note that the case where each \( j_i \) is scheduled on \( M_i \) corresponds to the negative decision, i.e., the situation where \( w \) is blocked on machines in \( Q_+ \). Whereas, each \( j_i \) being scheduled on \( M_{i-1} \) corresponds to the positive decision, i.e., window \( w \) is blocked on machines in \( Q_- \). \( \square \)

When the decision gadget is used in a hardness construction, we plug intervals to its slots. Then, the above lemma ensures that either all intervals placed in positive slots of the gadget can be scheduled and those placed on
the negative slots are blocked, or vice versa. Now, with both the blocking gadget and the decision gadgets defined, we show the following theorem.

**Theorem 1.9.** The problem of deciding whether there exists a complete schedule for INTERVAL SELECTION with cores is NP-complete even if all intervals have unit length.

**Proof.** The problem is obviously in NP. To show NP-hardness, we present a reduction from \((\leq 3,3)\)-SAT.

**Construction.** Let \(\Phi\) be an arbitrary instance of \((\leq 3,3)\)-SAT given by a set of clauses \(C = \{c_1, c_2, \ldots, c_r\}\) over a set of Boolean variables \(X = \{x_1, x_2, \ldots, x_s\}\), where each variable appears in \(\Phi\) at most three times, once as a negative literal and once or twice as a positive one.

We construct an instance \(S\) of the scheduling problem as follows (cf. Figure 1.4 along with the construction). We introduce three machines for each variable \(x_i\), denoted by \(M_{x_i,1}\), \(M_{x_i,2}\), and \(M_{x_i,3}\). On the machines we consider a window of four time units and denote the unit windows constituting it by \(w_1, w_2, w_3, w_4\). We introduce jobs as follows, using unit length intervals only. We place two blocking gadgets spanning all machines, one on the window \(w_1\) and the other on \(w_4\). For each variable \(x_i\), we place...
a decision gadget $D_{x_i}$ on machines $M_{x_i,1}, M_{x_i,2}, M_{x_i,3}$, such that $D_{x_i}$ has positive slots on $w_2$ on $M_{x_i,1}, M_{x_i,2}$, and a negative slot on $w_2$ on $M_{x_i,3}$ (i.e., $Q_+ = \{M_{x_i,1}, M_{x_i,2}\}$, $Q_- = \{M_{x_i,3}\}$) and occupies the windows $w_2$ and $w_3$. Recall that each decision gadget requires a blocking gadget—we use the blocking gadget on $w_4$ for this purpose. We place the intervals of $D_{x_i}$ that need to be blocked in such a way that they cover three quarters of the window $w_3$ and one quarter of $w_4$. By this, we achieve that they are never selected in any complete schedule and, at the same time, all the intervals for a single job of $D_{x_i}$ have a window in common (the second quarter of $w_3$), and thus they have a core. The decision of $D_{x_i}$ in a complete schedule will correspond to a truth assignment to variable $x_i$: A positive decision will correspond to a TRUE assignment and a negative decision will correspond to a FALSE assignment. Note that the positive/negative decision of $D_{x_i}$ is independent of the decisions of the other gadgets constructed the same way. Finally, we introduce a clause job $\beta_{c_j}$ for each clause $c_j$. Recall that each variable appears in $\Phi$ at most three times, once as a negative literal and once or twice as a positive literal. For each appearance of a variable $x_i$ as a positive literal in $c_j$ we place an interval $\beta_{c_j}$ on an unoccupied positive slot of the gadget $D_{x_i}$. Similarly, if $x_i$ appears in $c_j'$ as a negative literal, we place an interval for $\beta_{c_j'}$ on the negative slot of $D_{x_i}$. Note that we can place the clause jobs on the slots so that no slot is used twice, since each $x_i$ appears at most twice as a positive literal and once as a negative literal and $D_{x_i}$ has two positive slots and one negative. We place all the remaining intervals for clause jobs in a way that they cover half of the window $w_1$ and half of $w_2$. Because of the blocking gadget at $w_1$, none of these intervals can be selected in any complete schedule. Observe also that each clause job has a core (the first half of $w_2$), all intervals have the same length, and that the construction can be done in polynomial time.

Correctness. First, suppose that there is a complete schedule $S$ for $\Phi$. We construct a satisfying truth assignment $A$ for $\Phi$ as follows. For each variable $x_i$ we look at the schedule for the corresponding decision gadget $D_{x_i}$: If it decides positively, we set the value of $x_i$ in $A$ to TRUE, otherwise to FALSE. We show that every clause $c_j$ of $\Phi$ is satisfied by the resulting assignment $A$. The completeness of $S$ ensures that $\beta_{c_j}$ is scheduled on some machine, say $M_{x_i,k}$. Either it is scheduled on one of the two positive slots of $D_{x_i}$ and $x_i$ appears in $c_j$ as a positive literal, or it is scheduled on the negative slot of $D_{x_i}$ and $x_i$ appears in $c_j$ as a negative literal. The former case implies a positive decision of the gadget $D_{x_i}$ and hence $x_i$ is TRUE in $A$ by construction. The latter case implies a negative decision of the gadget $D_{x_i}$ and $x_i$ being set to FALSE in $A$. In both cases the clause $c_j$
is satisfied by $x_i$ in $A$.

Conversely, given a truth assignment $A$ that satisfies $\Phi$, we construct a complete schedule for $S$ as follows. We schedule all the jobs of the blocking gadgets in any order. We schedule all the jobs constituting a decision gadget $D_{x_i}$ so that if the variable $x_i$ is TRUE in $A$, we make a positive decision for $D_{x_i}$, and if $x_i$ is FALSE in $A$, we make a negative decision for $D_{x_i}$. We schedule the clause job for each clause $c_j$ as follows. Since $A$ satisfies $\Phi$, $c_j$ is satisfied by some variable $x_i$. Either $x_i$ appears in $c_j$ as a positive literal, in which case we schedule $\beta_{c_j}$ on its positive slot of $D_{x_i}$, or it appears as a negative literal, in which case we schedule it on the negative slot of $D_{x_i}$. In the former case $x_i$ is set to TRUE in $A$ which implies the positive decision of $D_{x_i}$, in the latter case $x_i$ is set to FALSE implying the negative decision of $D_{x_i}$. In both cases, $\beta_{c_j}$ can be scheduled on the specified slot of $D_{x_i}$ without overlapping with the scheduled jobs comprising $D_{x_i}$. Since no two scheduled clause jobs overlap, we can schedule them all in this way and obtain a complete schedule for $S$.  

1.3 Approximating Unit Interval Selection with Cores beyond 1/2

In this section we show that INTERVAL SELECTION with cores and unit intervals can be deterministically approximated strictly better than 1/2, without any restriction on the number of machines $m$ of the given instance. The results of this section were developed together with Enrico Kravina and Matúš Mihalák. Parts of the text in this section are based on or taken from a part of Master thesis of Enrico Kravina [52], which I co-supervised.

**Theorem 1.10.** There is a deterministic $\frac{501}{1000}$-approximation algorithm for INTERVAL SELECTION with cores and unit intervals.

The approximation ratio of our algorithm is at least $\frac{501}{1000}$. We believe that this ratio can be further improved by fine-tuning the parameters of the algorithm, and using a more careful analysis; none the less, we mainly aim to show that it beats the barrier of 1/2, and we try to keep the algorithm and its analysis relatively simple.

Recall, we use the term window to refer to a time interval independent of machines and jobs. If a job has more cores, we fix any of those (say, the left-most), and refer to it as the core of the job.
Chapter 1. Interval Selection

Figure 1.5: An instance of INTERVALSELECTION with cores and unit intervals with four machines. There are 8 jobs: \(a_1, a_2, a_3, a_4, b_1, b_2, b_3, b_4\). The labels of the intervals denote the identity of the underlying job. The generalization to any number of machines is straightforward. On \(m\) machines, the optimum of the instance has size \(2m - 1\). Both the multi-machine greedy and the scanline-greedy yield a solution of size \(m\). This implies that their approximation ratio is at most \(\frac{m}{2m-1}\), which tends to \(1/2\) as \(m\) grows.

1.3.1 Standard techniques fail

Before we describe the algorithm, we discuss some standard techniques – greedy approach and a shifting technique – and explain where they fail in the goal of achieving an approximation ratio better than \(1/2\).

Greedy is not better than \(1/2\)-approximation

To see that the multi-machine greedy does not achieve a better approximation ratio than \(1/2\), consider the instance of our problem shown in Figure 1.5. Actually, also another natural greedy algorithm fails on this instance: the scanline greedy algorithm scans all machines at once, from left to right, and always schedules the first available interval found by the scan.

Shifting technique

Because the intervals of a single job are placed “around” the core of the job, one may be tempted to use the standard shifting technique of separating the instance into many independent instances of small width (the x-coordinate distance between the leftmost and the rightmost cores):
Fix an optimum solution OPT for our problem, and consider a set of vertical lines equidistantly separated by a window of size $w \geq 1$; For any two neighboring lines, consider all intervals that lie entirely between the two lines as a new INTERVAL SELECTION subproblem. Obviously, no two intervals from two different subproblems may belong to the same job, and therefore, solutions of the two subproblems are disjoint, and the problems can thus be solved independently.

Moreover, there exists a placement of these lines (i.e., determining the $x$-coordinate of one single line determines the exact placement of all lines) such that at most $|OPT|/w$ many intervals from OPT intersect one of the lines. Then, discarding these intervals from OPT provides a solution $OPT'$ of size at least $(1 - 1/w) |OPT|$. Obviously, this is a lower bound on the overall quality of an optimum solution to all the subproblems. If we could solve the subproblems optimally for, say, $w = (3 - \varepsilon)$, one could achieve a $2/3 - \varepsilon'$-approximation for the overall problem. The problem with this approach is that the subproblems remain NP-hard (cf. Theorem 1.9) even for constant width of $w \geq 2$. For $w < 2$, any optimum solution for the subproblem places at most one interval on every machine (there is simply no space for two unit intervals in a window of width $w < 2$), and thus this problem can be solved in polynomial time by a reduction to maximum matching (match jobs/intervals with machines). Nonetheless, the quality of such a solution is only $(1 - 1/w) |OPT| \leq 1/2 \cdot |OPT|$, which does not beat the barrier of 1/2. For $w = 3 - \varepsilon$, each subproblem can be seen as a set packing problem with sets of the form $\{M_i, j_1, j_2\}$ and $\{M_i, j_1\}$ denoting that we can schedule job $j_1$ and job $j_2$ on machine $M_i$ (i.e., the two corresponding intervals of the jobs do not intersect). For this problem, only $(2/3 + \varepsilon)$-approximation algorithm is known [45], and thus the overall approximation guarantee becomes (ignoring the $\varepsilon$-terms) $2/3 \cdot (1 - 1/w) |OPT| \leq 4/9 \cdot |OPT|$, i.e., the approximation ratio guarantee becomes worse than 1/2 (which is achievable by a simple greedy algorithm).

### 1.3.2 The approximation algorithm

The approximation algorithm, which we call **SplitAndMerge**, is a recursive divide-and-conquer algorithm. Described on a high level, in every step it either provides a good enough solution for the considered (sub)instance (and goes back in the recursion), or identifies a middle subinstance – a set of jobs with cores inside a small window $W$, for which a good enough solution can be provided; in this case, the algorithm recursively proceeds with the
two subinstances induced by the jobs having cores left of $W$, and right of $W$, respectively, and then merges the three solutions. Here, good enough solution needs to schedule close-to-optimum many jobs of the subinstance, and at the same time, it must allow to merge the solutions from the left, middle, and the right subinstances without losing much of the quality.

The algorithm considers two types of a middle subinstance. The first type is induced by a window $W$ of size 6 which contains at most $m$ cores. The second type is structurally more involved: the algorithm first runs a greedy algorithm, called adaptive greedy, for the considered (sub)instance; then, if there is a window $W$ of size 20 such that the adaptive greedy schedules all jobs with cores in $W$, then one can identify a sub-window of $W$ that induces the middle instance.

The algorithm takes special care of middle windows $W$ for which the left or the right subinstance is empty. For this reason, we say that a window $W$ is on the left border of an instance, if there is no core strictly to the left of $W$, and at the same time there is a core at the left endpoint of $W$. We analogously define $W$ to be on the right border. We say that $W$ is in the interior of an instance if there is both a core strictly to the left and strictly to the right of $W$.

The algorithm SplitAndMerge works as follows (the auxiliary procedures are subroutines which are described in detail later):

1. If there are at most $m$ jobs, return a schedule where every job is scheduled (one on each of $m$ machines).

2a. Else if there is a window of size 6 in the interior of the instance that contains at most $m$ cores, then return the result obtained by the sparse interior procedure.

2b. Else if one of the two windows of size 3 on the borders of the instance contains at most $m$ cores, then return the result obtained by the sparse border procedure.

3. Else, run the adaptive greedy algorithm on the whole instance to obtain a solution $S_{AG}$; and mark all the cores of jobs which are not scheduled in $S_{AG}$.

   (Ia) If there is a window of size 20 in the interior of the instance which does not contain a marked core, then return the result obtained by the middle splitting procedure.

   (Ib) Else if one of the windows of size 9 on the borders of the instance does not contain a marked core, then return the result
obtained by the border splitting procedure.

(II) Else return the solution $S_{AG}$

We now describe the adaptive greedy algorithm: it consists of $m$ iterations, where $m$ is the number of machines. In each iteration it processes one machine by running the single-machine greedy on it. All jobs scheduled on that machine are made unavailable for subsequent iterations. The order in which the machines are processed is decided in an adaptive (greedy) way. In each iteration, the chosen machine is the one that maximizes the number of selected intervals (among all not yet processed machines). Such a machine is found by running single-machine greedy on each of the remaining unprocessed machines. Obviously, the running time of multi-machine greedy is polynomial in $n$ and $m$, since it runs the single-machine greedy $(m + (m - 1) + \ldots + 2) = O(m^2)$ many times.

The adaptive greedy algorithm has the following guarantee on the number of scheduled jobs:

Lemma 1.11. During the first $q$ iterations, the adaptive greedy selects at least $\left(\frac{q}{m} - \frac{q^2}{2m^2}\right)$ $s$ intervals, where $s$ is the optimum of the instance.

Proof. Let us fix an underlying optimum of size $s$. Let $t_i$, $i \in \{1, \ldots, m\}$, denote the number of intervals selected by adaptive greedy during its $i$th iteration and let $s_i$, $i \in \{1, \ldots, m\}$, denote the number of intervals of the underlying optimum that are still available in the beginning of the $i$th iteration.

Observe that an interval $I$ from the optimum can become unavailable for two reasons: either the adaptive greedy selects another interval $I'$ on the same machine that intersects $I$ and that ends earlier than $I$, or the adaptive greedy scheduled the job of $I$ earlier (by selecting an interval of this job on a different machine). Thus, every chosen interval of the adaptive greedy can make at most two intervals of the optimum unavailable. Therefore, if $t_i$ intervals are selected by the adaptive greedy in iteration $i$, we have $s_{i+1} \geq s_i - 2t_i$. In the beginning of the $i$th iteration, there are still $m - i + 1$ machines to be processed, and there are still $s_i$ intervals of the optimum available on those machines. Therefore, there is a machine with at least $\frac{s_i}{m-i+1}$ many intervals of the optimum and from the strategy of adaptive greedy and optimality of single-machine greedy it follows that $t_i \geq \frac{s_i}{m-i+1}$.

Using these recurrence relations we prove the lemma by induction on $m$ and $q \leq m$. We note that for $q = 1$ and arbitrary $m$, the statement of the
lemma holds, since in the first iteration at least \( \frac{s}{m} \) intervals are selected. Now let us consider the base case, that is, \( m = 1 \). Since \( q \leq m \), we are in the situation where \( q = 1 \) and the statement of the lemma follows from the just noted fact.

Next, assuming \( m \geq q \geq 2 \) and that the statement holds for any instance with \( m - 1 \) machines, we analyze the number of intervals selected in the first \( q \) iterations. After the fist iteration, \( t_1 \geq \frac{q}{m} \) intervals are selected and the optimum on the remaining machines is of size \( s_2 \geq s_1 - 2t_1 \). By applying the induction hypothesis we obtain that in the next \( q - 1 \) iterations at least \( \left( \frac{q-1}{m-1} - \frac{(q-1)^2}{2(m-1)^2} \right) (s_1 - 2t_1) \) intervals are selected. Therefore, altogether, in the next \( q \) iterations, at least \( t_1 + \left( \frac{q-1}{m-1} - \frac{(q-1)^2}{2(m-1)^2} \right) (s_1 - 2t_1) \) intervals are selected. For increasing values of \( t_1 \geq s_1/m \), this lower bound also increases, and thus the bound is minimized for \( t_1 = s_1/m \).

Therefore, at least

\[
\frac{s_1}{m} + \left( \frac{q-1}{m-1} - \frac{(q-1)^2}{2(m-1)^2} \right) (s_1 - 2\frac{s_1}{m})
\]

intervals are selected.

We conclude the proof by showing that \( \frac{s_1}{m} + \left( s_1 - 2\frac{s_1}{m} \right) \left( \frac{q-1}{m-1} - \frac{(q-1)^2}{2(m-1)^2} \right) \geq \left( \frac{q}{m} - \frac{q^2}{2m^2} \right) s_1 \) as follows. By performing a division by \( s_1 \) and expanding terms we obtain the equivalent inequality \( \frac{1}{m} + \left( \frac{q-1}{m-1} - \frac{(q-1)^2}{2(m-1)^2} \right) - \frac{2}{m} \left( \frac{q-1}{m-1} - \frac{(q-1)^2}{2(m-1)^2} \right) \geq \frac{q}{m} - \frac{q^2}{2m^2} \). Putting all terms on the left side we obtain

\[
\frac{1-q}{m} + \frac{q-1}{m-1} - \frac{(q-1)^2}{2(m-1)^2} - \frac{2(q-1)}{m(m-1)} + \frac{(q-1)^2}{m(m-1)^2} + \frac{q^2}{2m^2} \geq 0.
\]

Next we multiply the inequality with \( 2m^2(m-1)^2 \). The expression on the left hand side becomes

\[
(1-q)2m(m-1)^2 + (q-1)2m^2(m-1) - (q-1)^2m^2 - 2(q-1)2m(m-1) + (q-1)^22m + q^2(m-1)^2.
\]

We expand everything to obtain

\[
2m^3 - 4m^2 + 2m - 2m^3q + 4m^2q - 2mq + 2qm^3 - 2qm^2 - 2m^3 + 2m^2 - q^2m^2 + 2qm^2 - m^2 - 4m^2q + 4mq + 4m^2 - 4m + 2mq^2 - 4mq + 2m + q^2m^2 - 2q^2m + q^2
\]

which can be simplified to \( m^2 - 2mq + q^2 = (m - q)^2 \geq 0 \). \( \square \)

Observe that for \( q = m \), the algorithm schedules at least \( \frac{1}{2} \) intervals. Thus, this algorithm alone does not guarantee a better approximation ratio than \( \frac{1}{2} \). However, observe that in the first rounds, the lemma guarantees that
the algorithm takes larger fractions of \( s \) than in the last rounds. Thus, if we had a good alternative bound on the number of intervals selected by the algorithm in its last rounds, we could obtain a better approximation ratio by simply summing the two different lower bounds on the number of selected intervals, i.e., the lower bound for the first \( q' \) rounds of the algorithm, plus the lower bound for the last \( m - q' \) rounds of the algorithm. Later on, we will provide exactly such a lower bound on the number of intervals selected in the last rounds of the algorithm.

The sparse interior procedure

If the sparse interior procedure is called, then there must be a window \( W \) of size 6 in the interior that contains at most \( m \) cores. Let \( x \) be the left endpoint of \( W \). The sparse interior procedure creates a left subinstance consisting of jobs with cores to the left of \( x + 2 \), and a right subinstance consisting of jobs with cores to the right of \( x + 4 \). After that, it recursively calls \texttt{SplitAndMerge} to obtain solutions to these two subinstances. Finally, the sparse interior procedure merges these two results as follows. We observe that intervals of jobs with cores to the left of \( x + 2 \) cannot intersect intervals of jobs with cores to the right of \( x + 4 \) (since all intervals have unit length). This implies that the sparse interior procedure can merge the left and right solutions without any conflicts. It remains to add to this result all the jobs with cores in the window from \( x + 2 \) to \( x + 4 \). The sparse interior procedure schedules these jobs on those machines where no job with core in the window from \( x \) to \( x + 2 \) and no job with core in the window from \( x + 4 \) to \( x + 6 \) has been scheduled. Since by assumption there are at most \( m \) jobs in the window from \( x \) to \( x + 6 \), this can always be done.

The sparse border procedure

The sparse border procedure is similar to the sparse interior procedure. We describe only the sparse border procedure for the right border, since the procedure for the left border is symmetric. Suppose that the window \( W \) of size 3 on the right border of the instance contains at most \( m \) cores, and let \( x \) be the left endpoint of \( W \). The sparse border procedure uses \texttt{SplitAndMerge} (recursively) to obtain a solution for the subinstance that consists of the jobs with cores to the left of \( x + 2 \). We observe that the jobs with cores in the window from \( x + 2 \) to \( x + 3 \) cannot conflict with jobs with cores to the left of \( x \). Since there are no more than \( m \) jobs with cores in the window from \( x \) to \( x + 3 \), the sparse border procedure can schedule
all the jobs with cores in the window from $x+2$ to $x+3$ without causing any conflict.

**The middle splitting procedure**

The middle splitting procedure is applied if there is a window $W$ of size 20 in the interior of the instance with no marked core inside $W$. This means that all jobs with core inside $W$ have been scheduled by the adaptive greedy (i.e. they are in the solution $S_{AG}$).

The procedure identifies a *middle window*, a sub-window of $W$, to naturally split the instance into a left, middle, and right instance. The procedure recursively calls $\text{SplitAndMerge}$ to obtain solutions for the left and right instances, and combines them with the “unblocked” intervals of $S_{AG}$ having core in the middle window.

To choose the middle window, we first subdivide the window $W$ in 10 windows of size 2, see Figure 1.6. Starting from the left, we call these windows $w_5^L, w_4^L, w_3^L, w_2^L, w_1^L, w_1^R, w_2^R, w_3^R, w_4^R, w_5^R$. For a window $w$ we denote with $|w|$ the number of cores inside of $w$. For $i \in \{1, 2, 3, 4, 5\}$, we let $r_i = |w_i^L| + |w_i^R|$ and define the windows $\sigma_i$ as $\sigma_i = \bigcup_{j=1}^i (w_j^L \cup w_j^R)$.

Finally, we let $\kappa_i := |\sigma_i|$ and observe that $\kappa_i = \sum_{j=1}^i r_j$. The middle window is chosen among the four candidates $\sigma_2, \sigma_3, \sigma_4,$ and $\sigma_5$. Motivated by the desired approximation guarantee, let $\alpha := \frac{501}{1000}$. For $i \in \{2, 3, 4\}$ we say that the window $\sigma_i$ is *valid* if $r_i \leq \frac{1-\alpha}{\alpha} \kappa_{i-1}$. If one of $\sigma_2, \sigma_3$ or $\sigma_4$ is valid, we choose it as the middle window (if more than one of is valid, any one of them can be chosen). If on the other hand none of these three windows is valid, we choose $\sigma_5$ as the middle window.
The border splitting procedure

The border splitting procedure is applied if there is a window $W$ of size 9 on the left or on the right border of the instance with no marked intervals in it (which means that all jobs whose core is inside $W$ have been scheduled by the adaptive greedy). We only describe the border splitting procedure for $W$ on the right border of the instance, since the case where $W$ lies on the left border of the instance is symmetric.

Let $x$ be the left border of $W$. We define four windows $w_1$, $w_2$, $w_3$, and $w_4$ as follows (cf. Figure 1.6). Window $w_1$ has size 3 and ranges from $x + 6$ to $x + 9$. The remaining windows have size 2. The window $w_2$ ranges from $x + 4$ to $x + 6$, $w_3$ from $x + 2$ to $x + 4$, and $w_4$ from $x$ to $x + 2$. For $i \in \{1, 2, 3, 4\}$ we let $r_i = |w_i|$, i.e., $r_i$ is the number of cores in the window $w_i$. For $i \in \{1, 2, 3, 4\}$, we define the windows $\sigma_i$ as $\sigma_i = \bigcup_{j=1}^{i} w_j$.

Finally, let $\kappa_i := |\sigma_i|$. Like in the middle splitting procedure, the border splitting procedure has to choose an appropriate border window among the three candidates $\sigma_2$, $\sigma_3$, and $\sigma_4$. Again, we use the constant $\alpha = \frac{501}{1000}$. For $i \in \{2, 3\}$ we say that the window $\sigma_i$ is valid if $r_i \leq \frac{1-\alpha}{\alpha} \kappa_{i-1}$. If one of $\sigma_2$ or $\sigma_3$ is valid, we choose it as the border window (if more than one of is valid, any one of them can be chosen). Otherwise we choose $\sigma_4$ as the border window.

The border window naturally splits the instance into the left subinstance and the right subinstance. Restricting $S_{AG}$ to jobs with core inside the border window gives a solution for the right instance (scheduling all jobs), and we obtain a solution for the left subinstance by recursively calling $\text{SplitAndMerge}$. Afterwards, the algorithm merges the two solutions by discarding intervals of the solution for the right instance that conflict with the solution for the left subinstance.

1.3.3 Analysis

We show that the approximation ratio of $\text{SplitAndMerge}$ is at least $\frac{501}{1000}$ using an induction on the size of the input instances. The base of the induction consists of instances with at most $m$ intervals which are solved exactly by $\text{SplitAndMerge}$. $\text{SplitAndMerge}$ recursively calls different procedures. In the following subsections we analyze each of the procedures.
Analysis of the sparse interior:border procedures

Both the sparse interior procedure and the sparse border procedure divide the instance into a “left”, “middle”, and “right” subinstances (where the left or the right subinstances may be empty), solve the left and the right subinstances recursively, combine the obtained solutions without any conflicts, and additionally schedules all jobs of the middle instance. By induction hypothesis, each of the solutions of the smaller parts achieves a ratio of $\frac{501}{1000}$ over the optimum on the corresponding part. Therefore, the combined solution achieves at least the same ratio.

Analysis of the middle splitting procedure

We analyze the middle splitting procedure. We distinguish two cases. Either one of the windows $\sigma_2$, $\sigma_3$ or $\sigma_4$ is valid, or none of them is valid and $\sigma_5$ is chosen as the middle window.

Case 1. Let $\sigma_k$, $k \in \{2, 3, 4\}$ be the chosen (valid) window. The chosen window induces a left, a middle, and a right subinstance. We show that the algorithm approximates all three subinstances with approximation ratio $\frac{501}{1000}$ and these three solutions can be merged without any conflict, which implies that the merged solution is also a $\frac{501}{1000}$-approximation.

The left and the right subinstances are approximated with ratio $\frac{501}{1000}$ by inductive hypothesis. We now analyze the approximation ratio of the solution for the middle subinstance. Since window $\sigma_k$ is valid, we know that $r_k \leq \frac{1-\alpha}{\alpha} \kappa_{k-1}$, which we can rewrite as $\alpha \leq \frac{\kappa_{k-1}}{r_k + \kappa_{k-1}}$. Recall that the optimum of the middle subinstance is $r_k + \kappa_{k-1}$. Also, since the intervals inside of $\sigma_{k-1}$ do not conflict with jobs of the left or right subinstance (because their cores are more than two units away), there are at least $\kappa_{k-1}$ intervals from the solution of the middle subinstance which are not discarded when merging the subinstances. Therefore, the middle subinstance is approximated with ratio at least $\alpha = \frac{501}{1000}$.

Case 2. Consider the case where none of the three candidate middle windows is valid. First we show that this implies $\kappa_4 > 5m$ as follows. The fact that every window of size 6 contains more than $m$ cores implies that $|w_L^L| + |w_L^R| > m$ and that $|w_L^L| + |w_R^L| + |w_R^R| > m$. Adding these two inequalities we conclude that $2r_1 + r_2 > 2m$. Furthermore, since the window $\sigma_2$ is not valid, we have that $r_2 > \frac{1-\alpha}{\alpha} \kappa_1 = \frac{1-\alpha}{\alpha} r_1$. The last two inequalities can be combined and solved for $r_1 + r_2$ (multiply the
second with $\alpha$ and add it to the first) to obtain $\kappa_2 = r_1 + r_2 > \frac{2}{1+\alpha} m$. Since also $\sigma_3$ and $\sigma_4$ are not valid, we obtain $\kappa_4 = (r_4 + \kappa_3) > (\frac{1-\alpha}{\alpha} \kappa_3 + \kappa_3) = (\frac{1-\alpha}{\alpha} + 1) (r_3 + \kappa_2) > (\frac{1-\alpha}{\alpha} + 1)^2 \kappa_2 > (\frac{1-\alpha}{\alpha} + 1)^2 \frac{2}{1+\alpha} m = \frac{2}{(\alpha+1)\alpha} m > 5m$.

Now, to bound the number of jobs with cores in $\sigma_5$ that “survive” the merging with left and right subinstance solutions, we make two observations. First, notice that on each machine at most four intervals of the middle instance conflict with the left or the right solution (two per merging side). Second, since none of the jobs with cores in $\sigma_4$ conflicts with the left or the right solution, all these $\kappa_4$ jobs are scheduled in the combined solution. Therefore, the combined solution contains at least $\max(r_5 - 4m, 0) + \kappa_4$ jobs with cores in $\sigma_5$.

Since the optimum for the middle subinstance is $r_5 + \kappa_4$, we obtain that the approximation ratio for the middle subinstance is at least $\frac{\max(r_5 - 4m, 0) + \kappa_4}{r_5 + \kappa_4}$.

To show that this ratio is greater than $\frac{501}{1000}$, we distinguish two cases. If $r_5 \leq 4m$, then $r_5 \leq \frac{4}{5} 5m \leq \frac{4}{5} \kappa_4$, and we obtain that the approximation ratio of the middle subinstance is at least $\frac{\kappa_4}{r_5 + \kappa_4} \geq \frac{\kappa_4}{\frac{4}{5} \kappa_4 + \kappa_4} = \frac{5}{9} > \frac{501}{1000}$. If $r_5 \geq 4m$, then the approximation ratio is at least $\frac{r_5 - 4m + \kappa_4}{r_5 + \kappa_4} = 1 - \frac{4m}{r_5 + \kappa_4} \geq 1 - \frac{4m}{4m + 5m} = \frac{5}{9} > \frac{501}{1000}$.

### Analysis of the border splitting procedure

The analysis of the border splitting procedure is analogous to the analysis of the middle splitting procedure. We give the analysis for the case that the window is on the right border of the instance. The case where the window is on the left border of the instance is symmetric.

We consider two cases. Either one of the windows $\sigma_2$ or $\sigma_3$ is valid, or none of them is valid.

Suppose that at least one of the windows $\sigma_2$ or $\sigma_3$ is valid, and that the border splitting procedure chooses the valid window $\sigma_k$, $k \in \{2, 3\}$, as the border window. This induces a left and a right subinstance. The left subinstance is approximated by $\text{SPLITANDMERGE}$ with ratio at least $\frac{501}{1000}$ (by induction hypothesis). It remains to analyze the approximation ratio obtained for the right subinstance. Since the window $\sigma_k$ is valid, we know that $r_k \leq \frac{1-\alpha}{\alpha} \kappa_{k-1}$, which we can rewrite as $\alpha \leq \frac{\kappa_{k-1}}{r_k + \kappa_{k-1}}$.

Notice that the local optimum of the right instance is $r_k + \kappa_{k-1}$. Also
observe that at least \( \kappa_{k-1} \) intervals from the solution of the right section are not discarded when merging, since the intervals of jobs with cores in \( \sigma_{k-1} \) do not conflict with jobs of the left subinstance (since the distance between the respective cores is at least two). Therefore the right subinstance is approximated with approximation ratio at least \( \alpha = \frac{501}{1000} \). Thus, after merging, the approximation ratio for the whole instance is at least \( \frac{501}{1000} \).

We now consider the other case when both \( \sigma_2 \) and \( \sigma_3 \) are not valid and the window \( \sigma_4 \) is used as the border window. Because the window of size 3 on the right border of the instance contains at least \( m \) cores, we know that \( \kappa_1 > m \) and we can compute \( \kappa_3 = (r_3 + \kappa_2) \geq (\frac{1-\alpha}{\alpha} \kappa_2 + \kappa_2) = (\frac{1-\alpha}{\alpha} + 1)\kappa_2 \geq (\frac{1-\alpha}{\alpha} + 1)(r_2 + \kappa_1) \geq (\frac{1-\alpha}{\alpha} + 1)(\frac{1-\alpha}{\alpha} \kappa_1 + \kappa_1) = (\frac{1-\alpha}{\alpha} + 1)^2 \kappa_1 \geq \frac{1}{\alpha^2} m > 3m.

To finish the argument one has to observe that when merging left and right results, we lose at most two intervals per machine at the merging point. Also, when merging, none of the \( \kappa_3 \) jobs with cores in the window \( \sigma_3 \) are ever discarded from the solution of the right subinstance. Therefore the number of intervals from the solution of the right subinstance that are not discarded is at least \( \max(r_4 - 2m, 0) + \kappa_3 \). Since the optimum for the right subinstance is \( r_4 + \kappa_3 \), we obtain that the approximation ratio of the right instance is at least \( \frac{\max(r_4 - 2m, 0) + \kappa_3}{r_4 + \kappa_3} \).

We do a case distinction to show that this quantity is greater than \( \frac{3}{5} \). If \( r_4 \leq 2m \) then \( r_4 \leq \frac{2}{3} 3m \leq \frac{2}{3} \kappa_3 \) and the approximation ratio becomes \( \frac{\kappa_3}{r_4 + \kappa_3} \geq \frac{3}{5} > \frac{501}{1000} \). If \( r_4 \geq 2m \), then the approximation ratio becomes \( \frac{r_4 - 2m + \kappa_3}{r_4 + \kappa_3} = 1 - \frac{2m}{r_4 + \kappa_3} \geq 1 - \frac{2m}{2m + 3m} = \frac{3}{5} > \frac{501}{1000} \).

**Analysis when no splitting is necessary**

We consider an instance which has been processed by adaptive greedy and which contains at least one marked core (that is, a core of a job that has not been scheduled). We start with the following observations.

**Lemma 1.12.** If there is a marked core at \( x \in \mathbb{R} \), then on every machine adaptive greedy has selected an interval completely contained in the window from \( x - 2 \) to \( x + 1 \).

**Proof.** Consider a fixed machine \( M \). At some point during adaptive greedy, a run of single-machine greedy is performed on \( M \). Furthermore, consider an unscheduled job \( J \) with its core at \( x \). On machine \( M \), the interval \( I \) of
J lies completely between $x - 1$ and $x + 1$. Since $I$ has not been selected by single-machine greedy, it means that an interval $I'$ has been selected instead. This interval $I'$ has its right endpoint in the interior of $I$. Hence $I'$ has to lie completely between $x - 2$ and $x + 1$.

**Lemma 1.13.** If there are $k$ marked cores such that the distance between any two of those cores is strictly greater than 2, then on each machine at least $k$ intervals are selected.

**Proof.** From Lemma 1.12 we know that for each unmarked core at $x$, an interval completely inside the window from $x - 2$ to $x + 1$ is selected. Now consider two cores, at $x_1$ and at $x_2 > x_1 + 2$. Since the overlap of the windows $(x_1 - 2, x_1 + 1)$ and $(x_2 - 2, x_2 + 1)$ is less than one unit, the intervals selected for the marked cores $x_1$ and $x_2$ must be distinct.

Thus, if there are many marked cores far away from each other, then the adaptive greedy scheduled many jobs. We define the width of an instance as the distance between the leftmost and the rightmost core.

**Lemma 1.14.** If all windows of size 20 in the interior of the instance and both windows of width 9 at the borders of the instance contain at least one marked core, then adaptive greedy selected on each machine at least $\omega + 2$ intervals, where $\omega$ is the width of the instance.

**Proof.** Along with the proof, see the illustration in Figure 1.7. Let $x_1 < x_2 < \cdots < x_k$, $k \in \mathbb{N}$ be the positions of the marked cores (there may be multiple cores in each of these positions). For $i \in \{1, 2, \ldots \}$ let $w_i$ be the window from $x_1 + 22(i - 1)$ to $x_1 + 22i$. We divide every window $w_i$ into a left part consisting of all the points to the left or on $x_1 + 22(i - 1) + 2$, and into a right part consisting of all the points strictly to the right of $x_1 + 22(i - 1) + 2$. Let $q$ be the smallest positive integer such that $x_k$ is in $w_q$. Since each window of size 20 in the interior contains at least one marked core, there must be a marked core in the right part of every window $w_j$. We now show that there are at least $\frac{\omega + 2}{22}$ marked cores with distance at least 2 from each other, which together with Lemma 1.13 concludes the proof. We distinguish two cases: either $x_k$ lies in the left part of $w_q$, or it lies in the right part of $w_q$.

We first consider the case where $x_k$ lies in the left part of $w_q$. We know that there is a marked core at $x_1$. Furthermore we know that there is at least one marked core in the right part of each of the windows $w_1, \ldots, w_{q-1}$. Hence in total there are at least $q$ marked cores with distance at least 2 from each
Let \( s \) be the size of an optimum solution. It is easy to see that the width of the instance is at least \( \frac{s}{m} - 2 \). Therefore we know from Lemma 1.14 that on each machine at least \( \frac{1}{22} \frac{s}{m} \) intervals are selected.

We now apply Lemma 1.11 for a carefully chosen number of iterations \( q \) of the adaptive greedy. For this purpose let \( \mu \) be the smallest nonnegative number such that \( \left( \frac{21}{22} + \mu \right)m \) is a natural number. Note that \( \mu < \frac{1}{m} \). Now, from Lemma 1.11 we know that in the first \( \left( \frac{21}{22} + \mu \right)m \) iterations of adaptive greedy, at least \( \left( \frac{21}{22} + \mu \right) - \frac{1}{2} \left( \frac{21}{22} + \mu \right)^2 \right) s = \left( \frac{483}{668} + \frac{1}{22} \mu - \frac{1}{2} \mu^2 \right) \) s intervals
are selected.

Since on each machine at least $\frac{s}{22m}$ intervals are selected, it follows that during the last $m - \left(\frac{21}{22} + \mu\right)m$ iterations of adaptive greedy, at least $\frac{1}{22} s m \left(m - \left(\frac{21}{22} + \mu\right)m\right) = s \left(\frac{2}{968} - \frac{1}{22}\mu\right)$ intervals are selected.

We sum these two quantities and obtain that in total at least $s \left(\frac{485}{968} - \frac{1}{2}\mu^2\right)$ intervals are selected. Since $\mu \leq \frac{1}{m}$, for large $m$ this converges towards $\frac{485}{968}s$. In particular, for $m \geq 150$ we obtain an approximation ratio of at least $\frac{501}{1000}$. It can be shown that for $m < 150$ the algorithm also achieves approximation ratio of $\frac{501}{1000}$. For simplicity, let us omit this here. For the sake of seeing that there is a $\frac{501}{1000}$-approximation algorithm, one can modify the presented algorithm in that it runs the exact (optimum) algorithm from the literature for every $m < 150$, and runs \textsc{SplitAndMerge} for $m \geq 150$.

1.4 Hardness of Interval Selection with Small Number of Machines

In this and the following section we consider the non-restricted \textsc{Interval-Selection}. We show that, in contrast to \textsc{IntervalSelection} with cores, the problem is NP-hard even if the number of machines is constant. In particular, we prove that deciding whether there is a complete schedule is NP-complete already for three machines. In contrast, the problem is polynomially solvable for two machines [46]. We show that the problem of maximizing the number of scheduled intervals, on the other hand, is NP-hard already for two machines (while polynomially solvable for one machine). Moreover, all these hardness results hold even when all intervals have the same length.

We believe that the techniques used in the proofs may be of independent interest. The decision gadgets capture the relation between a schedule and an assignment. However, we also use properties of edge coloring that provide us with a mapping that lets us put the pieces together and finalize the construction of a scheduling problem under the required, rather restrictive conditions.

We also note that based on our construction, Kravina [52] showed \textsc{Interval-Selection} with 2 machines to be APX-hard. Nevertheless, for simplicity
of exposition, in this work we present our original NP-hardness proofs.

### 1.4.1 Unit Interval Selection with Three Machines.

We consider INTERVAL SELECTION with three machines and unit length intervals, with the objective of deciding whether there is a complete schedule. We will present a reduction from \((≤3,3)\)-SAT. To simplify the construction, we will use the blocking gadgets as defined in Section 1.2.1 and the decision gadgets as defined in Section 1.2.2.

**Lemma 1.15.** Let \(\Phi\) be an instance of \((≤3,3)\)-SAT, given by a set of clauses \(C\) over a set of Boolean variables \(X\). Then, there exists a mapping \(p\) from \(E = \{(x, c) \in X \times C \mid x \in c\}\) to the set \(\{M_1, M_2, M_3\}\), such that \(p(x, c) \neq p(x, c')\) for \(c \neq c'\) and \(p(x, c) \neq p(x', c)\) for \(x \neq x'\). Moreover, such a mapping \(p\) can be found in polynomial time.

**Proof.** We prove the statement by edge-coloring the bipartite graph \(G = (X \cup C, E)\). The structure of \((≤3,3)\)-SAT implies that all vertices of the constructed graph \(G\) have a degree at most 3. A bipartite graph is \(\Delta\)-edge-colorable in polynomial time, where \(\Delta\) is the maximum degree [49]. Therefore, the graph \(G\) is 3-edge-colorable, with colors from \(\{M_1, M_2, M_3\}\). This coloring gives us the desired mapping from \(E\) to \(\{M_1, M_2, M_3\}\).

**Theorem 1.16.** The problem of deciding whether there exists a complete schedule in INTERVAL SELECTION is NP-complete even for three machines and unit length intervals.

**Proof.** The problem is obviously in NP. To show the hardness, we reduce \((≤3,3)\)-SAT to it.

**Construction.** Let \(\Phi\) be an instance of \((≤3,3)\)-SAT, given by a set of clauses \(C = \{c_1, c_2, \ldots, c_r\}\) over a set of Boolean variables \(X = \{x_1, x_2, \ldots, x_s\}\). We construct from \(\Phi\) the following instance \(S\) of the INTERVAL SELECTION problem (cf. Figure 1.8 along with the construction), using three machines \(M_1, M_2, M_3\). We use a window of \(2s + 1\) units, and denote the unit windows constituting it by \(w_0, w_{1,1}, w_{1,2}, w_{2,1}, w_{2,2}, \ldots, w_{s,1}, w_{s,2}\). We introduce jobs with unit length intervals as follows. We place a blocking gadget on window \(w_0\) over all machines. For each variable \(x_i\) we place a decision gadget \(D_{x_i}\) on the machines such that it has two positive and one negative slot on window \(w_{i,1}\), in an arrangement that we will specify later. The gadget \(D_{x_i}\) occupies windows \(w_{i,1}\) and \(w_{i,2}\) and uses internally the blocking gadget on \(w_0\). The positive/negative decision of gadget \(D_{x_i}\)
corresponds to the truth assignment of the variable \(x_i\) and the decision of \(D_{x_i}\) is independent of the other decision gadgets. We introduce a clause job \(\beta_{c_j}\) for each clause \(c_j\). To place the intervals for \(\beta_{c_j}\), we look at the literals that appear in \(c_j\). For each appearance of a positive literal of some variable \(x_i\) in \(c_j\) we place an interval for \(\beta_{c_j}\) on a positive slot of \(D_{x_i}\), and for each appearance of a negative literal of \(x_i\) in \(c_j\) we place an interval for \(\beta_{c_j}\) on the negative slot of \(D_{x_i}\). If \(c_j\) contains only two literals, we place one interval for \(\beta_{c_j}\) on the window \(w_0\) so that it intersects the blocking gadget and cannot be selected in any complete schedule.

To obtain a valid construction, we need to ensure that all the intervals for each clause job \(\beta_{c_j}\) are placed on different machines, and at the same time, we require that each positive/negative slot of the decision gadgets is occupied by at most one interval. We now explain the exact placement of the positive/negative slots, as well as the distribution of the clause jobs over the slots that achieve this. We have three machines and we need to place each decision gadget so that it has its negative slot on some machine and its positive slots on the other two machines. Finding a way to arrange the decision gadgets and distribute their slots is equivalent to finding a mapping from a set of pairs (variable \(x\), clause \(c\) containing \(x\)) to the set \(\{M_1, M_2, M_3\}\) that assigns different machines to the variables in each clause and different machines to the clauses containing a fixed variable. Such a mapping can be efficiently constructed due to Lemma 1.15.

**Correctness.** The correctness of the construction can be showed by a similar argument as in the proof of Theorem 1.9. In short, first we suppose that there is a complete schedule \(S\) for \(S\) and we derive the truth assignment for each variable from the decisions of the corresponding decision gadget. Our construction ensures that this assignment satisfies every clause, since the corresponding clause job needs to be scheduled in \(S\). Conversely,
we explain how to construct a complete schedule $S$ from a satisfying truth assignment $A$: First we schedule the jobs of the blocking gadget in any order. Then we schedule the jobs constituting each decision gadget according to the truth assignment of the corresponding variable in $A$. Finally, for each clause, we pick one literal that satisfies it in $A$ and schedule its clause job on the corresponding slot of a decision gadget. By construction, no two intervals of clause jobs placed on slots of decision gadgets overlap (Lemma 1.15), hence all clause jobs can be scheduled in this way.

### 1.4.2 Unit Interval Selection with Two Machines.

We consider INTERVAL SELECTION with two machines and unit intervals and show that the problem of maximizing the number of scheduled intervals is NP-hard. The proof is similar to that of Theorem 1.16, but uses a reduction from (2,3)-MaxSat. To simplify the construction, we will again use the blocking gadget as defined in Section 1.2.1 and we will redefine the decision gadget, defined in Section 1.2.2. We state the following lemma, similar to Lemma 1.15.

**Lemma 1.17.** Let $\Phi$ be an instance of (2,3)-MaxSat, given by a set of clauses $C$ over a set of Boolean variables $X$. Then, there exists a mapping $p$ from $E = \{(x,c) \in X \times C \mid x \text{ appears in } c \text{ as a positive literal}\}$ to the set of machines $\{M_1, M_2\}$, such that $p(x,c) \neq p(x,c')$ for $c \neq c'$ and $p(x,c) \neq p(x',c)$ for $x \neq x'$. Moreover, such a mapping $p$ can be found in polynomial time.

**Proof.** As in the proof of Lemma 1.15, it is enough to realize that bipartite graph $G = (X \cup C, E)$ is 2-edge-colorable. $\square$

**Theorem 1.18.** Maximizing the number of scheduled intervals in INTERVAL SELECTION is NP-hard, even for two machines and unit length intervals.

**Proof.** To show the hardness of this INTERVAL SELECTION problem, we provide a reduction from (2,3)-MaxSat. The construction is similar to the construction for Theorem 1.16, but we use refined decision gadgets.

**Construction.** Let $\Phi$ be an instance of (2,3)-MaxSat, given by a set of clauses $C = \{c_1, c_2, \ldots, c_r\}$ over a set of Boolean variables $X = \{x_1, x_2, \ldots, x_s\}$. We construct from $\Phi$ the following instance $S$ of the INTERVAL SELECTION with two machines $M_1$ and $M_2$ (consider Figure 1.9 along with the construction). On the machines we consider a window of 3s units and we denote the unit
Before we introduce the jobs, we refine the decision gadgets that we use in the construction. Each decision gadget is for the two machines $M_1, M_2$, has two positive slots on some window and, additionally, two negative slots on another window, in such a way that if the jobs of the gadget are scheduled, either both positive slots are blocked and the negative slots are free, or vice versa. More precisely, we use a decision gadget $D$ for machines $M_1, M_2$ with two positive slots on some window $w$ and introduce two negative slots on both machines on the unit length window $w'$ that begins half a unit after the end of $w$ (see Figure 1.10). Recall that a decision gadget is made up of two jobs $j_0$ and $j_1$ with an interval for $j_0$ placed on $M_1$ so that its left end is at the center of $w$ and an interval for $j_1$ on $M_1$ placed so that its left end touches the right end of $w$. The intervals for $j_0$ and $j_1$ on $M_2$ are in the opposite arrangement. Therefore, if both $j_0$ and $j_1$ are scheduled, either $w$ is blocked on both machines and $w'$ is free (negative decision) or $w$ is free on both machines and $w'$ is blocked (positive decision).

We proceed with the actual construction. For each variable $x_i$ we place a refined decision gadget $D_{x_i}$ on the machines $M_1, M_2$ with positive slots on window $w_{i,1}$ and negative slots on the unit window consisting of the second half of $w_{i,2}$ and the first half of $w_{i,3}$. Again, a positive/negative decision of the gadget $D_{x_i}$ is in correspondence with a truth assignment of the variable $x_i$, and the decisions of different decision gadgets do not interfere.
For each clause $c_j$ we introduce a clause job $\beta_{c_j}$. In order to place intervals for clause jobs, we first look at all positive literals that appear in $\Phi$. For each $x_i$ that appears in $c_j$ as a positive literal, we place an interval for $\beta_{c_j}$ on a positive slot of $D_{x_i}$. For now, we assume that we can place clause jobs on positive slots in a way that no two intervals for a clause job are placed on the same machine and that at most one interval is placed on on each positive slot. If $x_i$ appears in $c_j$ as a negative literal, we place an interval for $\beta_{c_j}$ on a negative slot of $D_{x_i}$ in such a way that the two intervals of a job $\beta_{c_j}$ are placed on different machines. This can be achieved since only exactly one negative slot is used in each gadget, and we are thus free to choose which machine to place the interval on.

We now show that the intervals for clause jobs can be placed on positive slots so that no clause job has two intervals on the same machine and no positive slot is occupied twice. This is equivalent to requiring that there is a mapping from $X \times C$ to the set $\{M_1, M_2\}$ that assigns different machines to two positive literals appearing in the same clause and assigns different machines to the same positive literals in different clauses. Lemma 1.17 ensures the existence of such a mapping. Thus, we can obtain a valid placement for interval of clause jobs such that no two of them overlap.

The above construction for an instance $S$ of INTERVALSELECTION has two machines and $2s + r$ jobs. All intervals have unit length and we can construct the presented reduction from (2,3)-MaxSat in linear time. We conclude the proof by showing that there is a schedule for $S$ that schedules at least $2s + k$ jobs if and only if there is a truth assignment for $\Phi$ that satisfies at least $k$ clauses.

**Correctness.** We can prove the correctness by a similar argument as in the proof of Theorem 1.16. First we suppose there is a maximum schedule $S$ for $S$ that schedules at least $2s + k$ jobs and we derive a truth assignment for $\Phi$ that satisfies at least $k$ clauses. As the first step, we show that there exists a schedule $S'$ that schedules also at least $2s + k$ jobs but in which all jobs of the decision gadgets are scheduled. Each interval of a job of a decision gadget intersects with exactly one positive/negative slot and therefore it intersects with at most one scheduled interval of a clause job in $S$. Therefore, we can modify $S$ by scheduling each not yet scheduled job of a decision gadget instead of a scheduled clause job. We obtain $S'$ that schedules all $2s$ jobs of decision gadgets and at least $k$ clause jobs. We construct a truth assignment $A$ of the variables in $\Phi$ according to the decisions of decision gadgets in $S'$. At least $k$ clauses are satisfied by $A$ in $\Phi$, due to the fact that $k$ clause jobs are scheduled in $S'$, with a similar
argument as in the proof of Theorem 1.16.

Conversely, we suppose there is a truth assignment $A$ that satisfies at least $k$ clauses of $\Phi$ and construct a schedule $S$ that schedules at least $2s + k$ jobs. We schedule the $2s$ decision gadgets' jobs according to the truth assignment of the variables in $A$. For each clause $c_j$ that is satisfied by $A$ we schedule the clause job $\beta_{c_j}$ on a slot of decision gadget that corresponds to a literal that satisfies the $c_j$ in $A$. The construction (together with Lemma 1.17) ensures that no two intervals of clause jobs overlap and thus at least $k$ clause jobs can be scheduled in this way. Therefore, we can construct a schedule $S$ that contains at least $2s + k$ jobs.

\[\square\]

\section{Approximating Interval Selection with Two Machines}

In this section we present a $2/3$-approximation algorithm for \textsc{Interval-Selection} with two machines. Recall that \textsc{Interval-Selection} on one machine is solvable by a single-machine greedy: a simple greedy algorithm that considers all intervals on the machine sorted by the right end-points in the ascending order and selects each considered interval if it does not intersect any of the previously selected intervals. For every selected interval, the corresponding job is scheduled. For short, let us denote this algorithm by $A$.

We can also apply the greedy algorithm $A$ in the setting with two machines $M_1$ and $M_2$. Let $A_{M_1 \rightarrow M_2}$ be the algorithm that first runs $A$ on machine $M_1$ and schedules on $M_1$ a set of jobs $S_1$, then removes from $M_2$ (the intervals for) the jobs in $S_1$, runs $A$ on machine $M_2$ and schedules on $M_2$ a set of jobs $S_{12}$. Thus, the algorithm returns a feasible schedule that consists of the set of jobs $S_1 \cup S_{12}$, with $S_1 \cap S_{12} = \emptyset$. This algorithm gives a $1/2$-approximation [64], which is tight for the algorithm.

Obviously, we can run the greedy algorithm in the other direction, i.e., first on $M_2$ and then on $M_1$ (denoted by $A_{M_2 \rightarrow M_1}$), which again gives a $1/2$-approximation. Perhaps surprisingly, the algorithm that chooses the better solution of the two provided by $A_{M_1 \rightarrow M_2}$ and $A_{M_2 \rightarrow M_1}$ is a $2/3$-approximation. Even though the algorithm, let us call it BestDirection, is extremely simple, the analysis thereof is more interesting.

We introduce the following notation. For a given instance $I$ of \textsc{Interval-Selection} on two machines and a subset of its jobs $S$, let us denote by $M_i(S)$
Algorithm 1 Approximation for IntervalSelection on two machines

Require: Instance $I$ of IntervalSelection on two machines $M_1$, $M_2$, consisting of a set of jobs $J$.

1: $S_1 \leftarrow$ jobs scheduled by running algorithm $A$ on $M_1(J)$
2: $S_{12} \leftarrow$ jobs scheduled by running algorithm $A$ on $M_2(J \setminus S_1)$
3: $S_2 \leftarrow$ jobs scheduled by running algorithm $A$ on $M_2(J)$
4: $S_{21} \leftarrow$ jobs scheduled by running algorithm $A$ on $M_1(J \setminus S_2)$
5: return the larger of the two schedules $S_1 \cup S_{12}$ and $S_2 \cup S_{21}$.

The intervals of $I$ corresponding to jobs in $S$ on machine $M_i$ (for $i \in \{1, 2\}$). Then, $M_i(S)$ can be also seen as an instance of IntervalSelection on one machine. With this notation, we summarize the algorithm BestDirection in Algorithm 1. Note that all the steps of the algorithm can be computed in polynomial time, since the greedy algorithm $A$ runs in polynomial time.

To analyze the performance of algorithm BestDirection, we relate the output of the algorithm, i.e., $\max(S_1 \cup S_{12}, S_2 \cup S_{21})$ to an optimum schedule $O = O_1 \cup O_2$, where $O_1$ and $O_2$ are sets of jobs corresponding to non-intersecting sets of intervals on $M_1$ and $M_2$, respectively, with $O_1 \cap O_2 = \emptyset$. Let $S_1, S_2, S_{12}, S_{21}$ be obtained in the course of algorithm BestDirection (as given in Algorithm 1). Since $A$ finds an optimum on a single machine, if follows that $|S_1| \geq |O_1|$ and $|S_2| \geq |O_2|$. Let $S_1^O$ be the subset of jobs of $S_1$ that are also in $O_1$, and let $S_1^R$ be the remaining jobs of $S_1$. Analogically, we subdivide $S_2$ into $S_2^O$ and $S_2^R$.

Next, we give a lower bounds on the numbers of jobs selected in the course of the algorithm.

**Lemma 1.19.** Algorithm $A$ schedules on $M_1(J \setminus S_2)$ at least $|O_1| - |S_2^R|$ jobs.

*Proof.* Since $O_1$ and $O_2$ are disjoint sets of jobs, the same is true for $O_1$ and $S_2^O$. Let us denote by $O_1^R$ those jobs that are in $O_1$ but not in $S_2$. Obviously, the set of jobs $O_1^R := O_1 \setminus S_2 = O_1 \setminus (S_2^O \cup S_2^R) = O_1 \setminus S_2^R$ is of size at least $|O_1| - |S_2^R|$, and it corresponds to a set of non-intersecting intervals in $M_1(J \setminus S_2)$. Thus, running algorithm $A$ on $M_1(J \setminus S_2)$ gives a schedule with at least $|O_1^R| \geq |O_1| - |S_2^R|$ scheduled jobs. \hfill $\Box$

**Lemma 1.20.** Algorithm $A$ schedules on $M_2(J \setminus S_1)$ at least $\frac{|O_2| - |S_1| + |S_2^R|}{2}$ jobs.

*Proof.* Consider the sets of jobs $X := O_2 \setminus S_1$ and $Y := S_2 \setminus S_1$, and note that each corresponds to a set of non-intersecting intervals in $M_2(J \setminus S_1)$. \hfill $\Box$
Theorem 1.21. Algorithm BestDirection is a $2/3$-approximation algorithm for IntervalSelection on two machines.

Proof. By Lemma 1.20, $|S_{12}| \geq \frac{1}{2}(|O_2| - |S_1| + |S_2^R|).$ Thus, $|S_1| + |S_{12}| \geq \frac{1}{2}(|S_1| + |O_2| + |S_2^R|) \geq \frac{1}{2}(|O| + |S_2^R|)$. On the other hand, by Lemma 1.19, $|S_{21}| \geq |O_1| - |S_2^R|$. Thus, $|S_2| + |S_{21}| \geq |S_2| + |O_1| - |S_2^R| \geq |O| - |S_2^R|$. Therefore, the algorithm produces a solution of size at least $\max(\frac{1}{2}(|O| + |S_2^R|), |O| - |S_2^R|)$, and this expression is minimized for $|S_2^R| = \frac{1}{3}|O|$, which gives $\frac{2}{3}|O|$. The analysis is tight, as the example from Figure 1.12 shows.

In the next chapter we will see that the simple idea behind this algorithm can be further generalized, not only for $m$ machines, giving an approximation ratio of $(\frac{1}{2} + \frac{1}{2m(m-1)})$, but also to be used on a much wider class of problems.
In this chapter, we identify several NP-hard maximization problems to be explicitly composed of simpler ones. We present a generic approximation framework to combine existing algorithms for these simpler problems to obtain a black-box approximation to the composed problems.

For intuition, consider a ground set of elements $S$, together with a "blue family" of its subsets $B \subseteq 2^S$ and a "red family" $R \subseteq 2^S$. Assume that both $(S, B)$ and $(S, R)$ forms each an independence system, i.e., $\emptyset \in B$ and if $B \in B$, then every subset $B' \subseteq B$ is in $B$ (the same for $R$). Consider the problem of finding two disjoint sets $B \in B$ and $R \in R$ such that the size of the union $B \cup R$ is maximized. More generally, instead of 2 colored families (blue and red), there can be $k$ colored families and the goal is to find $k$ pairwise-disjoint sets, one of each color, such that the size of their union is maximized. It is easy to imagine that instead of the explicitly given colored families we have $k$ maximization problems (defined over a common ground set) such that the feasible solutions of each of these problems define a family of one color. Such problems can for example ask for matchings, or cliques in a graph, or feasible solutions for a Knapsack instance. Then the $k$-composed problem asks for $k$ pairwise-disjoint sets, one feasible solution for each of the given problems, such that the size of their union is maximized. This problem is especially interesting when composed of different problems, or of different instances of the same problem, since then their sets of feasible solutions differ. Many optimization problems can be formulated as a $k$-composed problem, including the maximum red-blue split subgraph; the interval selection problem; and the maximum
edge-$k$-coloring. Unfortunately, $k$-composed problems are often NP-hard even when the individual maximization problems are polynomially-time solvable and $k = 2$.

There is a large amount of research concerning maximization problems of similar flavor. Closely related areas include assignment problems, welfare maximization and combinatorial auctions, and coverage problems. Some of the previously studied problems contain our problem as a special case, and some of these problems are a special case of our problem (e.g., the maximum coverage problem). We postpone the detailed discussion of these problems to the section on related work (Section 2.4), but we point out that, in many cases, randomized $(1 - 1/e)$-approximation algorithms are known. These state-of-the-art algorithms are based on solving LP relaxations of the IP formulations of the problems, and subsequently using randomized rounding techniques. In most cases, no derandomizations of the algorithms are known, and often stated as open problems. Interestingly, if we focus on combinatorial and deterministic algorithms, in most cases the best known such algorithms are greedy algorithms giving “only” a 1/2-approximation.

In this chapter, we present a black-box approximation framework for approximating $k$-composed problem $P^k$, composed of maximization problems $P_1, \ldots, P_k$, that uses existing exact or approximation algorithms for $P_1, \ldots, P_k$. Specifically, given a $\rho_i$-approximation algorithm for $P_i$, we get a $(2/3) \cdot \min_i \rho_i$-approximation algorithm for $P^2$, and a $(\frac{1}{2} + \frac{1}{2k(k-1)}) \cdot \min_i \rho_i$-approximation algorithm for $P^k$, $k \geq 3$. For a constant $k$, the running time overhead of our framework is only a constant multiplicative factor. Even though the strategy proposed in this thesis does not attain the ratio of the randomized algorithms, it beats the known greedy algorithms and still stays extremely simple, purely combinatorial and deterministic.

We first explain our framework using a concrete graph-theoretic maximization problem – the maximum red-blue split subgraph problem (Section 2.1), which is a generalization of INTERVALSELECTION seen in Chapter 1. We define the general class of $k$-composed (maximization) problems and show that many previously studied problems fit into this framework (Section 2.2). We then give a black-box greedy algorithm for 2-composed problems and show that it is a $(2/3) \cdot \min_i \rho_i$-approximation algorithm (Section 2.3.1). We generalize the black-box greedy algorithm for $k$-composed problems in Section 2.3.2, we show that for $k = 3$ the algorithm gives a $(7/12) \cdot \min_i \rho_i$-approximation, and by bootstrapping the approximability analysis for $k'$-composed problems, $k' < k$, we obtain a lower bound of $(\frac{1}{2} + \frac{1}{2k(k-1)}) \cdot \min_i \rho_i$ on the approximation ratio of the algorithm for
Approximating a Maximum Red-Blue Split Subgraph

$k$-composed problems. In Section 2.3.3 we give upper bounds on the approximation ratio of the algorithm. We postpone the discussion on related work to the end of the chapter (Section 2.4).

The results in this chapter are a joint work with my co-authors. In particular, the lower bound on the approximation ratio of the algorithm for $k = 3$ (Theorem 2.11) as well as the upper bound on the approximation ratio for instances of INTERVAL SELECTION (Theorem 2.15) are a joint work with Enrico Kravina and Matúš Mihalák. These results were already presented in a similar way in the setting of INTERVAL SELECTION in Enrico Kravina’s Master thesis [52]. The remaining results in this chapter, including the generalization of the algorithm beyond INTERVAL SELECTION and deriving lower and upper bounds on the approximation ratio of the algorithm in this general case are joint work with Matúš Mihalák. The results presented in this chapter have been submitted for publication [12].

2.1 Approximating a Maximum Red-Blue Split Subgraph

A red-blue graph $G = (V, E = R \cup B)$ is an undirected graph with the vertex set $V$ and an edge set $E$ such that each edge $e \in E$ is either blue ($e \in B$), red ($e \in R$), or both. We call $G_R = (V, R)$ the red graph, and $G_B = (V, B)$ the blue graph. A red-blue graph is a red/blue-split graph (or simply R/B-split graph) if $V$ can be partitioned into two (disjoint) sets such that one is an independent set in $G_R$, and the other one is an independent set in $G_B$. The MAX-R/B-SPLIT GRAPH problem asks, for a given red-blue graph $G$, for a largest subset $V'$ of $V$ such that the graph $G[V']$, the subgraph of $G$ induced by $V'$, is an R/B-split graph.

Since INDEPENDENT SET, i.e., the problem of finding a maximum independent set in a graph, is NP-hard to approximate within $n^{1-\varepsilon}$ [43], MAX-R/B-SPLIT GRAPH has for general graphs at least this complexity. Faigle et al. [31] showed that even if both $G_R$ and $G_B$ are comparability graphs, MAX-R/B-SPLIT GRAPH cannot be approximated better than $31/32$, unless P=NP. Since INDEPENDENT SET on comparability graphs can be solved in polynomial time, Faigle et al. [31] proposed a simple 1/2-approximation algorithm for the case when both $G_R$ and $G_B$ are comparability graphs: Compute a maximum independent set $MIS_R$ in $G_R$, and a maximum independent set $MIS_B$ in $G_B$, and return the larger of the two. Obviously, an optimum solution to MAX-R/B-SPLIT GRAPH returns a vertex set $V' \subseteq V$
Algorithm 2 Approximation algorithm for Max R/B-split subgraph

Require: R/B-graph $G = (V, R \cup B)$
1: $S_R \leftarrow \text{max independent set in } G_R$
2: $S_{RB} \leftarrow \text{max independent set in } G_B[V \setminus S_R], \text{subgraph of } G_B \text{ induced by } V \setminus S_R.$
3: $S_B \leftarrow \text{max independent set in } G_B$
4: $S_{BR} \leftarrow \text{max independent set in } G_R[V \setminus S_B], \text{subgraph of } G_R \text{ induced by } V \setminus S_B.$
5: return the larger of the two sets $S_R \cup S_{RB}$ and $S_B \cup S_{BR}.$

such that $V'$ can be partitioned into an independent set $V'_R$ in $G_R$ and an independent set $V'_B$ in $G_B$ (with $|V'| = |V'_R| + |V'_B|$). It follows that $|V'_B| \leq |\text{MIS}_B|$ and $|V'_R| \leq |\text{MIS}_R|$, which gives the claimed approximation ratio. Improving this approximation ratio is posed as an open question in subsequent work of Korach et al. [50], which study the subgraph characterization of R/B-split graphs and their special case, König Egerváry graphs.

In the following, we give a $2/3$-approximation algorithm for MAX-R/B-SPLITGRAPH for the case when both $G_R$ and $G_B$ belong to a class of graphs closed under taking subgraphs, and for which INDEPENDENTSET is solvable in polynomial time. This, in particular, implies a $2/3$-approximation algorithm for MAX-R/B-SPLITGRAPH with both $G_R$ and $G_B$ being comparability graphs.

First, let us note that INTERVALSELECTION on two machines (defined in Chapter 1) can be seen as MAX-R/B-SPLITGRAPH for the case when both $G_R$ and $G_B$ are interval graphs as follows. Each job corresponds to a vertex, and the intervals on one machine determine the edges of one color. In particular, for each pair of jobs whose intervals intersect on machine $M_1$, there is a red edge between the corresponding vertices. Similarly, there is a blue edge for every intersecting pair on $M_2$. Then, a non-intersecting set of intervals corresponds to an independent set. Clearly, a solution to INTERVALSELECTION is feasible if and only if the corresponding solution to MAX-R/B-SPLITGRAPH is also feasible. Thus, we can directly translate algorithm BESTDIRECTION to the graph theory language as summarized in Algorithm 2.

Note that all steps of Algorithm 2 can be computed in polynomial time whenever both $G_R$ and $G_B$ belong to a class of graphs such that every induced subgraph again belongs to the same class, and INDEPENDENTSET is solvable in polynomial time. In particular, these conditions are satisfied.
Composed optimization problems

by both comparability graphs and interval graphs. Using this observation and following the exact same analysis as in Section 1.5, and only slightly changing the notation and terminology we obtain the following.

**Theorem 2.1.** Let $\mathcal{G}$ be a class of graphs closed under taking induced subgraphs, and for which INDEPENDENTSET is solvable in polynomial time. Then, Algorithm 2 is a $2/3$-approximation algorithm for MAX-R/B-SPLITGRAPH on $\mathcal{G}$.

As we will soon see, Algorithm 2 together with the result in Theorem 2.1 can be generalized even further, suggesting a versatile strategy that can be applied to a much broader class of problems.

### 2.2 Composed optimization problems

MAX-R/B-SPLITGRAPH can be seen as the problem of finding two disjoint independent sets $V_R$ and $V_B$ over the same set of vertices $V$, but in two different graphs $G_R = (V, E_R)$ and $G_B = (V, E_B)$, respectively. We now generalize these ideas. Recall, an NP-maximization problem $P$ has input instances $I$, a set of feasible solutions $F(I)$ for every instance $I \in I$, a size function $size : F(I) \rightarrow \mathbb{R}$ for every $I \in I$, and the goal is to find a feasible set of maximum size (cf. [54]).

**Definition 2.2 (Independence system).** An independence system $I = (S, \mathcal{F})$ is a pair, where $S$ is a finite set, and $\mathcal{F} \subseteq 2^S$ is a non-empty family of subsets of $S$ closed under taking subsets, i.e., $\emptyset \in \mathcal{F}$, and if $X \subseteq Y \in \mathcal{F}$, then $X \in \mathcal{F}$. We call $S$ the ground set of $I$, and elements of $\mathcal{F}$ the feasible sets/solutions of $I$.

**Definition 2.3 (Maximization problem over independence systems).** A maximization problem over independence systems is a maximization problem where the feasible solutions of every input form an independence system $I = (S, \mathcal{F})$, where every element $s \in S$ is assigned a weight $w(s)$\(^1\), and where the size of a solution $F \subseteq S$ is defined as $w(F) = \sum_{e \in F} w(e)$.

**Definition 2.4 (Composed maximization problem).** Let $P_1, P_2, \ldots, P_k$ be maximization problems over independence systems. A $k$-composed (maximization) problem from problems $P_1, \ldots, P_k$, denoted as $P^k(P_1, \ldots, P_k)$, is a maximization problem where every instance $I$ of $P^k$ is given by (i) $k$ independence systems $I_j = (S_j, \mathcal{F}_j)$, $j = 1, \ldots, k$, where every $I_j$ is an input for problem $P_j$, and (ii) one weight function $w : S_1 \cup \ldots \cup S_k \rightarrow \mathbb{R}$. The set of

\(^1\)If in every instance of the maximization problem $P$ we have $w(s) = w(s')$ for every $s, s' \in S$, we say that problem $P$ is unweighted.
feasible solutions $\mathcal{F}$ of instance $I$ is the set of $k$-tuples \((F_1, F_2, \ldots, F_k) \mid F_i \in \mathcal{F}_i \text{ for any } i, F_i \cap F_j = \emptyset\)^2. The size of a feasible solution $F = (F_1, F_2, \ldots, F_k)$ is $w(F) = \sum_{e \in \bigcup_i F_i} w(e) = w(F_1) + \cdots + w(F_k)$.

We remark that a $k$-composed problem is in fact a maximization problem over a union of independence systems, where a feasible set is explicitly partitioned into feasible sets of the $k$ original independence systems. In most problems, the independence systems $I_j$ are not given explicitly by listing all feasible sets (it would then be easy to find an optimum solution in linear time), but rather implicitly (in a compressed form), as we demonstrate next.

### 2.2.1 Examples of composed optimization problems

**Max-R/B-SplitGraph.** Obviously, `INDEPENDENTSET` can be seen as a maximization problem over an independence system – the ground set is the set of vertices and the feasible sets are the independent sets of the given graph. Then, **Max-R/B-SplitGraph** is an unweighted 2-composed problem \(P^2\) from problems \(P_1\) and \(P_2\), where \(P_1\) and \(P_2\) correspond to the problem `INDEPENDENTSET`, whose two instances play the role of the input graphs \(G_R\) and \(G_B\). We note that the obvious generalization of **Max-R/B-SplitGraph** to \(k \geq 3\) input graphs may also be of interest.

**IntervalSelection.** Similarly, also `INTERVALSELECTION` is a \(k\)-composed problem \(P^k(P_1, \ldots, P_k)\) where each individual problem \(P_i\) corresponds to `INDEPENDENTSET` in an interval graph induced by the intervals on machine \(M_i\).

Further generalization is a variant of `INTERVALSELECTION` where on any machine, the selected intervals can intersect, but at most \(r\) at any time point. This variant arises naturally in scheduling where multiple resources of one type are available. On a single machine, this variant is solvable in polynomial time [68]. On \(k\) machines, this variant can again be seen as a \(k\)-composed problem.

**Max-R/B-Matching.** An independence system may also be defined over the set of edges of a graph. For example, given a red/blue graph \(G = (V, E = E_R \cup E_B)\), the problem **Max-R/B-Matching** asks for a maximum

---

2Whenever $\mathcal{F}_i$ are downward-closed systems, we can give up the requirement $F_i \cap F_j = \emptyset$: We can iteratively for every pair with $F_i \cap F_j \neq \emptyset$ substitute $F_j$ with $F'_j := F_j \setminus F_i$ and obtain a feasible solution of the same size. For problems where $\mathcal{F}$ are not downward-closed, this requirement is crucial.
cardinality subset \( F \subseteq E \) of edges so that \( F \) can be divided into two parts \( F_R \) and \( F_B \), \( F = F_R \cup F_B \), so that \( F_R \subseteq E_R \) and \( F_B \subseteq E_B \) are two matchings in \( G \). The problem can again be easily generalized to be composed of \( k \geq 2 \) graphs \( G_i(V,E_i) \), \( i = 1, \ldots, k \). We note that if all the \( k \) edge sets \( E_i \) are the same, the problem has been studied under the name Max-Edge-k-Coloring [33].

**Composed Paths.** Finding a longest \( s-t \) path problem can be viewed as a maximization problem on independence systems by defining the family of feasible solutions to be the set of all sub-paths (i.e., edges) of all \( s-t \) paths. The \( k \)-composed problem (using the same edge set \( E \)) then follows immediately: find \( k \) \( s-t \) paths \( P_1, \ldots, P_k \) maximizing \( \sum_{e \in P_1 \cup \cdots \cup P_k} w(e) \).

**Composing non-graph problems.** Maximization problems over independence systems can be also found outside of the algorithmic graph theory. The problem of finding a maximum increasing subsequence in a sequence is such a maximization problem. Then, the corresponding \( k \)-composed problem asks, for a given sequence, for \( k \) disjoint increasing subsequences having the maximum number of elements. The notion of independence is also present in linear systems. Given a set of linear inequalities \( Ax \leq b \), the problem of finding the maximum number of satisfiable inequalities is a maximization problem on independence systems. The \( k \)-composed problem of \( k \) such individual problems \( Ax_i \leq b_i \) then asks for \( k \) disjoint sets \( A_1, \ldots, A_k \) of “rows” of \( A \), such that the inequalities corresponding to \( A_i \) are satisfiable in \( Ax_i \leq b_i \), and \( |A_1 \cup \cdots \cup A_k| \) is maximized.

**Composed Knapsack.** The classic Knapsack problem can be seen as maximization over independent systems: the feasible sets are exactly those sets of items that satisfy the given capacity constraint. The Knapsack problem can be used to create the following composed problem. Consider a set of items \( X \), where each item \( x_i \in X \) has a value \( a_i \), a weight \( w_i \), and a volume \( v_i \). We have two knapsacks available - one that has a maximum volume capacity \( V \), but unlimited weight capacity, and a second one that has a limited weight capacity \( W \), but unlimited volume capacity. The goal is to find a subset of items in \( X \) of maximum total value that can be packed into the two backpacks, given the respective constraints. The generalization to \( k \) Knapsacks is obvious.

**Composing different problems.** In all the above examples, the problems \( P_1, P_2, \ldots, P_k \) (that compose the problem \( P^k \)) are the same. Our framework is not limited to such composed problems, and in fact, one can consider all kinds of crazy combinations of \( P_i \)s and \( P_j \)s. To name one, \( P^2 \) can be composed of the maximum matching problem, and of the maximum forest
problem: Given two graphs $G_1 = (V, E_1)$ and $G_2 = (V, E_2)$, problem $P_1$ asks for a maximum matching in $G_1$, whereas problem $P_2$ asks for a forest of maximum size in $G_2$. Here, the set of edges $E_1 \cup E_2$ forms the ground set of $P_2$, and the two problems are competing over $E_1 \cup E_2$, as each edge can be taken only by one of them.

Complexity of composed problems

Sadly, many of the composed problems are NP-hard even if composed from two polynomially solvable problems (as the first two examples show). It is a natural question to study the approximability of the composed problems, and, especially, to quantify how the approximability of the underlying problems $P_1$ and $P_2$ influences the approximability of the composed problem $P_2$. In this chapter, we make a step in this direction.

2.3 Approximating composed problems

Consider a $k$-composed maximization problem $P_k(P_1, \ldots, P_k)$. Let $A_i$ be a $\rho_i$-approximation algorithm for $P_i$. Consider the following algorithm $A_g$ for $P_k$. Given an instance $I = (S_1 \cup \ldots \cup S_k, w, (I_1, \ldots, I_k))$ of $P_k$, $I_i = (S_i, F_i)$, compute for every $i$ a solution $F_i \in F_i$ by running algorithm $A_i$ on the input $(S_i, F_i)$; return the set $F_i$ of the largest weight $w(F_i)$ among the computed sets. We have seen that when $k = 2$ and $\rho_1 = \rho_2 = 1$, the algorithm gives a $1/2$-approximation (cf. the introduction in Section 2.1). The same arguments show the following.

Proposition 2.5. Algorithm $A_g$ is a $(\frac{1}{k} \cdot \min_i \rho_i)$-approximation algorithm for $P_k$.

In the following, we provide an algorithm that achieves a better approximation ratio. We first analyze the case $k = 2$, before discussing the general case.

2.3.1 Approximating 2-composed maximization problems

Algorithm 3 is a generalization of the 2/3-approximation algorithm for Max-R/B-SplitGraph. Here, for $X \subseteq S_i$, the notation $I_i[S_i \setminus X]$ denotes the instance of $P_i$ where the weight of each element of $X$ is set to zero.
Algorithm 3 Approximation algorithm for 2-composed problem $P^2$

**Require:** An instance $I = (S_1 \cup \cdots \cup S_k, w, I_1 = (S_1, F_1), I_2 = (S_2, F_2))$ of $P^2(P_1, P_2)$, together with approximation algorithms $A_1, A_2$ for the respective problems.

1. $F_1 \leftarrow$ feasible set in $I_1$ obtained by $A_1$
2. $F_{12} \leftarrow$ feasible set in $I_2[S \setminus F_1]$ obtained by $A_2$
3. $F_2 \leftarrow$ feasible set in $F_2$ obtained by $A_2$
4. $F_{21} \leftarrow$ feasible set $I_1[S \setminus F_2]$ obtained by $A_1$
5. **return** from the solutions $(F_1, F_{12})$ and $(F_2, F_{21})$ the one of larger weight.

(Seen alternatively, elements of $X$ are removed from every set in $F_i$.) Note that Algorithm 3 is well defined and runs in polynomial time. The analysis of the approximation guarantee follows the lines of the analysis of Algorithm 2, but needs additional twists, as we are now dealing with weighted problems and approximation algorithms $A_1$ and $A_2$ (instead of non-weighted problems and exact algorithms as before).

Let $I = (S, w, (I_1, I_2))$ be an instance of $P^2$, and consider an optimum solution $O = O_1 \cup O_2$ where $O_1$ is a feasible solution for $I_1$, $O_2$ is a feasible solution for $I_2$, and $O_1 \cap O_2 = \emptyset$. Let $F_1, F_2, F_{12}, F_{21}$ be obtained in the course of Algorithm 3 as described in the pseudocode. As before, we subdivide $F_1$ into disjoint sets $F_1^O = F_1 \cap O_1$, and $F_1^R$. Analogously, we define $F_2^O$ and $F_2^R$, $F_2 = F_2^O \cup F_2^R$.

**Lemma 2.6.** After the set $F_2$ is selected, there is a feasible solution for the instance $I_1[S \setminus F_2]$ of weight at least $w(O_1) - w(F_2^R)$.

**Proof.** Since $O_2$ and $O_1$ are disjoint, so are $F_2^O$ and $O_1$. Obviously, $O_1^R = O_1 \setminus F_2 = O_1 \setminus (F_2^O \cup F_2^R) = O_1 \setminus F_2^R$ is a feasible solution for $I_1$ of weight at least $w(O_1) - w(F_2^R)$. \hfill \Box

**Lemma 2.7.** After the set $F_1$ is selected, there is a feasible solution for the instance $I_2[S \setminus F_1]$ of weight at least $\frac{w(O_2) - w(F_1) + w(F_2^R)}{2}$.

**Proof.** Obviously, $X := O_2 \setminus F_1$ and $Y := F_2 \setminus F_1$ are two (potentially identical) feasible solutions for the instance $I_2[S \setminus F_1]$. Let us set $r := w(O_2 \cap F_1 \cap F_2)$, $s := w(F_2 \cap (F_1 \setminus O_2))$, and $t := w(O_2 \cap (F_1 \setminus F_2))$. Obviously, $w(X) = w(O_2) - r - t$. Similarly, $w(Y) = w(F_2) - r - s$.

The better of the two solutions has a weight at least $\frac{w(X) + w(Y)}{2} = \frac{1}{2}(w(O_2) + w(F_2) - 2r - s - t) \geq \frac{1}{2}(w(O_2) + w(F_2) - w(F_1) - w(F_2^O)) = \frac{1}{2}(w(O_2) - w(F_2^O))$. \hfill \Box
We remark that with a more careful analysis, it can be shown that the algorithm in fact produces a solution of weight at least $\frac{2}{3}(\rho_1 w(O_1) + \rho_2 w(O_2))$. 

**Theorem 2.8.** Algorithm 3 is a $(2/3) \cdot \min(\rho_1, \rho_2)$-approximation algorithm.

**Proof.** Let $\rho = \min(\rho_1, \rho_2)$. Thus, $w(F_1) \geq \rho w(O_1)$ and $w(F_2) \geq \rho w(O_2)$. By Lemma 2.7, $w(F_{12}) \geq \rho_2 \frac{1}{2}(w(O_2) - w(F_1) + w(F_R^2)) \geq \rho \frac{1}{2}(w(O_2) - w(F_1) + w(F_R^2))$. Thus, $w(F_1) + w(F_{12}) \geq w(F_1)(1 - \rho/2) + \rho_2 \frac{1}{2}(w(O_2) + w(F_R^2)) \geq \rho \frac{1}{2}(w(F_1) + w(O_2) + w(F_R^2)) \geq \rho \frac{1}{2}(w(O) + w(F_R^2))$.

On the other hand, by Lemma 2.6, $w(F_{21}) \geq \rho_1 (w(O_1) - w(F_R^2)) \geq \rho(w(O_1) - w(F_R^2))$. Thus, $w(F_2) + w(F_{21}) \geq \rho w(O_2) + \rho w(O_1) - \rho w(F_R^2) \geq \rho(w(O) - w(F_R^2))$. Therefore, the algorithm produces a solution of size at least

$$\max \left( \rho \frac{1}{2} \left( w(O) + w(F_R^2) \right), \rho \left( w(O) - w(F_R^2) \right) \right).$$

This expression is minimized for $w(F_R^2) = \frac{1}{3}w(O)$, which gives $\rho_2 \frac{1}{3}w(O)$. Thus, Algorithm 3 is a $\rho_2 \frac{1}{3}$-approximation algorithm. 

We remark that with a more careful analysis, it can be shown that the algorithm in fact produces a solution of weight at least $\frac{2}{3}(\rho_1 w(O_1) + \rho_2 w(O_2))$. 

---

**Algorithm 4 Approximation algorithm for $k$-composed problem**

**Require:** an instance $I = (S, w, (I_1, \ldots, I_k))$ of a $k$-composed problem $P^k(P_1, \ldots, P_k)$ together with approximation algorithms $A_1, \ldots, A_k$ for the respective problems.

1: for $\pi \leftarrow$ permutation of $\{1, \ldots, k\}$ do
2: \hspace{1em} $\text{Result}_\pi \leftarrow \emptyset$; \hspace{1em} $S' \leftarrow S$
3: \hspace{1em} for $i \leftarrow 1, \ldots, k$ do
4: \hspace{2em} $F_{\pi(i)} \leftarrow$ feasible set in $I_{\pi(i)}[S']$ obtained by $A_{\pi(i)}$
5: \hspace{1em} $\text{Result}_\pi \leftarrow \text{Result}_\pi \cup F_{\pi(i)}$; \hspace{1em} $S' \leftarrow S' \setminus F_{\pi(i)}$
6: \hspace{1em} end for
7: end for
8: return among the sets $\text{Result}_\pi$, $\pi$ permutation of $\{1, \ldots, k\}$, select the one with the greatest cardinality

$$w(F_1) + w(F_R^2),$$

where the inequality follows from the facts that $r + s + t \leq w(F_1)$ and $r \leq w(F_R^2)$. 

\hfill $\square$
2.3.2 Approximating $k$-composed optimization problems

We can naturally extend Algorithm 3 for $k$-composed problems: we consider all permutations of $(P_1, \ldots, P_k)$ and solve these problems in the imposed order by applying existing approximation algorithms $A_1, A_2, \ldots, A_k$, and take the best obtained result (see Algorithm 4). We refer to this algorithm as the general black-box algorithm. We note that due to the nature of trying all permutations of $k$ elements, the algorithm is not polynomial in $k$.

Assuming each $A_i$ is a $\rho_i$-approximation algorithm for $P_i$, we derive lower bounds on the approximation ratio achieved by the general black-box algorithm. In particular, first we show that the general black-box algorithm gives a $(7/12) \cdot \min(\rho_1, \rho_2, \rho_3)$-approximation for the 3-composed problem $P_3$, and this ratio is tight for the algorithm. Then, using the approximation ratio obtained for $k = 3$, we show in an inductive way that for $k \geq 3$ the algorithm achieves at least a $(1/2 + \frac{1}{2k(k-1)}) \cdot \rho$-approximation for the problem $P_k$, $k$-composed of $\rho$-approximable problems.

One of the difficulties in analyzing the general black-box algorithm for general $k$ is that it considers all possible orders in which to solve the induced instances of $I_i$, for $i = 1, \ldots, k$, and takes the best of all the obtained results. We overcome this difficulty by analyzing not the best outcome (obtained from a specific order), but the average performance (obtained over all $k!$ orders).

Given a $k$-composed problem $P_k(P_1, \ldots, P_k)$, we construct an auxiliary $k$-composed problem $\bar{P}_k(\bar{P}_1, \ldots, \bar{P}_k)$, for which the general black-box algorithm performs the same for every considered order of the problems $\bar{P}_1, \ldots, \bar{P}_k$. From the performance of the algorithm on any order of the problems in $\bar{P}_k$ we then obtain a lower bound on the performance of the algorithm on the “best” order in the original $k$-composed problem $P_k$. For simplicity of exposition, we assume that for every instance of $P_k$,
We now have the key ingredient to prove the following theorem. We define

\[ \text{Theorem 2.11} \]

were developed together with Enrico Kravina and Matúš

**Definition 2.9** (Homogeneous Problem). Let \( P_k(P_1, \ldots, P_k) \) be a \( k \)-composed problem. We define the homogeneous problem \( \tilde{P}_k(\tilde{P}_1, \ldots, \tilde{P}_k) \) of the problem \( P_k \) by its input instances \( \tilde{I} \). The set of instances \( \tilde{I} \) of \( \tilde{P}_k \) is induced by the instances \( I \) of \( P_k \). In particular, for each instance \( I = (S, w, I_1 = (S, \mathcal{F}_1), \ldots, I_k = (S, \mathcal{F}_k)) \) of the (original) \( k \)-composed problem \( P_k \), the set \( \tilde{I} \) contains an instance \( \text{hom}(I) = (S^*, w^*, (S^*, \mathcal{F}^*), \ldots, (S^*, \mathcal{F}^*_k)) \) obtained as follows. Let \( \{I_{\pi_1}, I_{\pi_2}, \ldots, I_{\pi_{k!}}\} \) denote all the \( k! \) orders of \( \{I_1 = (S, \mathcal{F}_1), \ldots, I_k = (S, \mathcal{F}_k)\} \), where \( I_{\pi} = (I_{\pi(1)}, I_{\pi(2)}, \ldots, I_{\pi(k)}) \). Let us denote the \( j \)th component of a tuple \( I_{\pi} \) by \( I_{\pi}^j \). The ground set \( S^* \) of \( \text{hom}(I) \) consists of \( k! \) independent (disjoint) copies of \( S_{\pi(1)}, S_{\pi(2)}, \ldots, S_{\pi(k!)} \), and, to simplify the notation, we imagine it as a \( k! \)-tuple \( (S_{\pi_1}, S_{\pi_2}, \ldots, S_{\pi_{k!}}) \). We define \( w^* \) as the natural extension of \( w \) (every copy of an element \( e \) has the same weight \( w(e) \) under \( w^* \)). Every \( I_{\pi}^j = I_{\pi(j)} = (S, \mathcal{F}_{\pi(j)}) \) naturally defines an instance of the problem \( P_{\pi(j)} \) over the ground set \( S_{\pi(j)} \). The component \( \tilde{I}_j = (S^*, \mathcal{F}^*_j) \), \( j = 1, \ldots, k \), of \( \text{hom}(I) \) consists of the “\( k! \) individual problems \( I_{\pi_1}^j, I_{\pi_2}^j, \ldots, I_{\pi_{k!}}^j \)”, i.e., for every \( \pi \), we transform a feasible set \( F \in \mathcal{F}_{\pi(j)} \) over elements in \( S \) into a feasible set \( F^* \in \mathcal{F}^* \) over the corresponding elements \( S_{\pi(j)} \). The set of feasible solutions of \( \tilde{I}_j \) is just the union (over all permutation) of the feasible solutions of the individual instances \( I_{\pi}^j \).

**Lemma 2.10.** Let \( I = (S, w, (S, \mathcal{F}_1), \ldots, (S, \mathcal{F}_k)) \) be an instance of a \( k \)-composed problem \( P_k \). The approximation ratio of the general black-box algorithm on \( I \) is at least as good as the approximation ratio of the algorithm on \( \text{hom}(I) \).

**Proof.** Let \( A_{\pi}(I) \) denote the elements that are picked by the general black-box algorithm when it considers the instances \( I_1, \ldots, I_k \) in the order \( I_{\pi(1)}, \ldots, I_{\pi(k)} \). It follows from the definition of \( \text{hom}(I) \) that the overall performance of the general black-box algorithm on \( \text{hom}(I) \) is \( \sum_{\pi} w(A_{\pi}(I)) \). Dividing this number by \( k! \), we get that the performance of the algorithm on \( \text{hom}(I) \) reflects an average performance of the general black-box algorithm when trying all \( k! \) orders in which it tries to solve the (sub)instances \( I_1, \ldots, I_k \). \( \square \)

We now have the key ingredient to prove the following theorem. We remark that the just presented idea of a homogeneous instance, as well as Theorem 2.11 were developed together with Enrico Kravina and Matúš
Theorem 2.11. The general black-box algorithm is a \((7/12) \cdot \min(\rho_1, \rho_2, \rho_3)\)-approximation algorithm for the 3-composed problem \(P^3\).

Proof. To prove the claimed lower bound for \(k = 3\), we try to bound the size of the obtained solution, by subdividing it into smaller sets and bounding their sizes with respect to the fractions of the optimum and other auxiliary sets. The obtained set of inequalities can be solved by an LP-solver and the duality-theorem.

Let \(\text{hom}(I) = (S, w, I_1, I_2, I_3)\) be a homogeneous instance. Let \(O = \{O_1, O_2, O_3\}\) be an optimum for the instance \(\text{hom}(I)\). Since \(\text{hom}(I)\) is homogeneous, it follows that \(w(O_1) = w(O_2) = w(O_3)\). Similarly, it follows that the algorithm performs equally well on every considered order \(\pi\). Thus, in the following we consider a fixed order \(\pi = (1, 2, 3)\) and analyze the weight of the solution obtained by the general black-box algorithm when considering the order \(\pi\). Let \(\rho = \min_i \rho_i\) be the minimum approximation ratio of the algorithms that compose the black-box algorithm. Let further \(F_1, F_2, F_3, F_{12}, F_{21}, F_{13}, F_{31}, F_{23}, F_{32}\), and \(F_\sigma, \sigma\) a permutation of \(\{1, 2, 3\}\), define the feasible sets chosen by the algorithm in a similar way as before. Again, it follows that \(w(F_1) = w(F_2) = w(F_3)\), and \(w(F_{ab}) = w(F_{cd})\), \(a, b, c, d, \in \{1, 2, 3\}\), \(a \neq b, c \neq d\), and \(w(F_\sigma) = w(F_\sigma')\) for any two permutations \(\sigma\) and \(\sigma'\) of \(\{1, 2, 3\}\). The output of the algorithm for the order \(\pi = (1, 2, 3)\) are the sets \(F_1, F_{12},\) and \(F_{123}\). We now focus on bounding the sizes of these and related sets.

Since the general black-box algorithm computes an optimum solution on \(I_1\), it follows that

\[
\frac{1}{\rho} w(F_1) \geq w(O_1).
\]  

(2.1)

At the same time, \(F_1\) contains some elements of \(O_1, O_2\) and \(O_3\), and therefore \(w(F_1) \geq \sum_i w(F_1 \cap O_i)\). We further refine this to obtain

\[
\frac{1}{\rho} w(F_1) \geq w(F_1) \geq \sum_i (w(O_i \cap F_1 \cap F_2 \cap F_3) + w(O_i \cap F_1 \cap F_2 \cap \bar{F}_3) + w(O_i \cap F_1 \cap \bar{F}_2 \cap \bar{F}_3) + w(O_i \cap \bar{F}_1 \cap \bar{F}_2 \cap \bar{F}_3)),
\]  

(2.2)

where \(\bar{F}_i\) stands for the complement of \(F_i\), i.e., \(\bar{F}_i := S \setminus F_i\).
Clearly, $F_{12}$ can still choose what remained from $O_2$ after $F_1$ was chosen, thus we get

$$
\frac{1}{\rho} w(F_{12}) \geq w(O_2) - w(O_2 \cap F_1 \cap F_2 \cap F_3) - w(O_2 \cap F_1 \cap F_2 \cap \bar{F}_3) - w(O_2 \cap F_1 \cap \bar{F}_2 \cap F_3) - w(O_i \cap F_1 \cap \bar{F}_2 \cap \bar{F}_3). \quad (2.3)
$$

Also, some elements of $F_{12}$ belong to $O_1$, $O_2$, and $O_3$. Thus,

$$
\frac{1}{\rho} w(F_{12}) \geq w(F_{12} \cap O_1) + w(F_{12} \cap O_2) + w(F_{12} \cap O_3). \quad (2.4)
$$

Clearly, $F_{123}$ can still choose from $O_3$ what remained after $F_1$ and $F_{12}$ were chosen, i.e., $w(F_{123}) \geq w(O_3) - w(O_3 \cap F_1) - w(O_3 \cap F_{12})$, which can be written as

$$
\frac{1}{\rho} w(F_{123}) \geq w(O_3) - w(O_3 \cap F_1 \cap F_2 \cap F_3) - w(O_3 \cap F_1 \cap F_2 \cap \bar{F}_3) - w(O_3 \cap F_1 \cap \bar{F}_2 \cap F_3) - w(O_3 \cap F_1 \cap \bar{F}_2 \cap \bar{F}_3) - w(O_3 \cap F_2). \quad (2.5)
$$

Also, $F_{123}$ could contain the elements of $O_1 \cap F_3$ and $O_2 \cap F_3$, if they were not chosen before by $F_1$ and $F_{12}$, i.e.,

$$
\frac{1}{\rho} w(F_{123}) \geq w(O_1 \cap F_1 \cap F_2 \cap F_3) + w(O_1 \cap F_1 \cap F_2 \cap \bar{F}_3) + w(O_2 \cap \bar{F}_1 \cap F_2 \cap F_3) + w(O_2 \cap \bar{F}_1 \cap \bar{F}_2 \cap F_3) - w(O_1 \cap F_3 \cap F_{12}) - w(O_2 \cap F_3 \cap F_{12}). \quad (2.6)
$$

Furthermore, $F_{123}$ can contain the elements of $O_2 \cap F_{13}$ that are not chosen by $F_{12}$, which implies that

$$
\frac{1}{\rho} w(F_{123}) \geq w(O_2 \cap F_{13}) - w(O_2 \cap F_{12}). \quad (2.7)
$$

The presented lower bounds on $w(F_1)$, $w(F_{12})$, and $w(F_{123})$ are expressed in fractions of the optimum $w(O_1)$, $w(O_2)$, and $w(O_3)$. In most of the inequalities, if there is a negative term (e.g., the last term $-w(O_3 \cap F_{12})$ in Eq. (2.5)), which could make the lower bound smaller, then there is a corresponding positive term in another equality (e.g., the first term...
Approximating composed problems

$w(O_2 \cap F_{13})$ in Eq. (2.7)$^3$, which then makes the respective lower bound larger. An exception is the inequality from Eq. (2.6), where the negative term $-w(O_2 \cap F_3 \cap F_{12})$ does not have a corresponding positive term.$^4$

Therefore, we need to bound the damage that such a negative term can have:

$$w(O_2 \cap F_3 \cap F_{12}) \leq w(O_2 \cap \bar{F}_1 \cap F_2 \cap F_3) + w(O_2 \cap \bar{F}_1 \cap \bar{F}_2 \cap F_3). \quad (2.8)$$

Apart from the listed inequalities, there are several trivial ones of the form $w(X) \geq w(X \cap Y)$ (where $X$ and $Y$ are any terms that were used so far in the inequalities; e.g., the inequality from Footnote 4). Moreover, there are also inequalities stemming from the property of homogeneous instances stating that $w(f(1,2,3)) = w(f(\pi(1),\pi(2),\pi(3)))$, where $f(1,2,3)$ is any term that was used before and using indices 1,2,3 (e.g., the equalities $w(F_1) = w(F_2) = w(F_3)$, or $w(O_1 \cap F_{23}) = w(O_2 \cap F_{13}) = w(O_3 \cap F_{12})$ are of this kind).

By putting all these inequalities together, one can show, e.g., by an LP-solver and the duality-theorem, that

$$\frac{1}{\rho} (w(F_1) + w(F_{12}) + w(F_{123})) \geq \frac{7}{12} (w(O_1) + w(O_2) + w(O_3)).$$

Thus, the general black-box algorithm for the 3-composed problem $P^3$ is a $(7\rho/12)$-approximation algorithm. $\square$

Clearly, similar strategy could be applied to find lower bounds on the approximation ratio of the algorithm for $k > 3$. However, with growing $k$, the number of cases that need to be considered grow quickly and we do not have a concise and elegant way to tackle this. Instead, using a simpler analysis we show that the general black-box algorithm gives an approximation ratio of $(1/2 + \frac{1}{2k(k-1)}) \cdot \min_i \rho_i$ for $k \geq 3$. This ratio is tight for the algorithm for $k = 3$ (see Section 2.3.3).

**Lemma 2.12.** Assume the general black-box algorithm is a $\xi_{k-1}$-approximation for the $(k-1)$-composed problem, $0 \leq \xi_{k-1} \leq 1$. Let $\rho \leq \min_i \rho_i$ be given. Then the algorithm achieves $\frac{(k-1-\rho)\cdot \xi_{k-1} + \rho}{k}$-approximation for the $k$-composed problem.

$^3$In a homogeneous instance, $w(O_2 \cap F_{13}) = w(O_3 \cap F_{12}) = w(O_1 \cap F_{23})$.

$^4$Observe that the second negative term $-w(O_1 \cap F_3 \cap F_{12})$ from Eq. (2.6) has a corresponding positive term $w(O_1 \cap F_{12}) \geq w(O_1 \cap F_3 \cap F_{12})$ in Eq. (2.4).
Theorem 2.13. The general black-box algorithm is a \(1/2 + \frac{1}{2k(k-1)}\) \(\rho\)-approximation algorithm for the problem \(P^k\), \(k \geq 3\), \(k\)-composed of \(\rho\)-approximable problems.

Proof. Follows from Lemma 2.12 and Theorem 2.11 by setting \(\xi_3 = (7/12) \cdot \min_{i} \rho_i\) (use induction and observe that \(\rho_i \cdot \rho_j \leq \min(\rho_i, \rho_j)\) whenever \(\rho_i, \rho_j \leq 1\)).

By applying the just presented general black-box algorithm on INTERVAL-SELECTION (discussed in Chapter 1), we obtain a deterministic algorithm which for the case with \(m \geq 3\) machines gives an approximation ratio of \((1/2 + \frac{1}{2m(m-1)})\), that is strictly greater than 1/2.

Clearly, this strategy can be applied to derive non-trivial approximation algorithms for many other \(k\)-composed problems (e.g., examples in Section 2.2.1).

2.3.3 Upper bounds on the approximation ratio of the algorithm

In this section we present instances on which the general black-box algorithm performs “poorly”, thus we bound from above the approximation ratio achieved by the algorithm. We have seen (Theorem 2.11) that the
algorithm gives a 7/12-approximation for 3-composed problem, and now Theorem 2.14 shows that this ratio is tight for the algorithm.

**Theorem 2.14.** The approximation ratio of the general black-box algorithm on k-composed problem for $k \geq 3$ is bounded from above by $\frac{7}{12}$.

**Proof.** Consider the following instance $I = (S_1 \cup S_2 \cup S_3, F_1, F_2, F_3)$ of $P^3$.

$S_1 = S_2 = S_3 = \{1, \ldots, 12\}$

$F_1 = \{\{1, 2, 3, 4\}, \{5, 6, 9, 10\}, \{7, 9\}, \{5, 11\}\}$

$F_2 = \{\{5, 6, 7, 8\}, \{1, 2, 9, 10\}, \{3, 9\}, \{1, 11\}\}$

$F_3 = \{\{9, 10, 11, 12\}, \{1, 2, 5, 6\}, \{3, 5\}, \{1, 7\}\}$

Assume each given algorithm $A_i$ for $P_i$ takes the largest available subset of $F_i$ and in case of a tie it always prefers the rightmost subset. Then, the optimum (in bold) is of size 12, but the solution obtained by the general black-box algorithm (underlined for the order 1,2,3) is of size 7 only (other orders give the same).

Note that the instance used in Theorem 2.14 may lack the structure required by a specific $k$-composed problem (e.g., IntervalSelection). In such a case, this bound does not apply. For IntervalSelection we prove the following upper bound. This construction was developed together with Enrico Kravina and Matúš Mihalák, and appeared in a similar fashion also in Kravina’s Master thesis [52].

**Theorem 2.15.** The approximation ratio of the general black-box algorithm on IntervalSelection with $m$ machines is bounded from above by

$$\frac{1}{(m-1)(m-1)!} + \sum_{k=0}^{m-1} \frac{1}{k!} - 1 \frac{m}{(m-1)(m-1)!} + \sum_{k=0}^{m-2} \frac{1}{k!}.$$

For $m = 2$, it gives $2/3$, and the instance corresponds to the one described in Section 1.5. For $m = 3$, it gives $7/11$. As $m$ grows, the expression converges to $\frac{e-1}{e}$.

**Proof.** Let us denote by $A$ the algorithm for IntervalSelection on a single machine that sorts the intervals on the machine by the end-points, and then greedily, from left to right, selects a maximum number of non-overlapping intervals. Recall that on a single machine, $A$ produces a solution of optimal size. Let us consider the general black-box algorithm to which we plug the
algorithm $A$ as the black-box algorithms $A_1, \ldots, A_m$. In the following, we give an IntervalSelection instance $I_m$ on which this general black-box algorithm performs “poorly”.

Let us first describe the intended structure and properties of $I_m$. It consists of $m$ machines $M_1, \ldots, M_m$. The intervals on the machines are arranged in a two layered structure, each interval is either in the upper layer or in the lower layer. Each interval in the upper layer is “paired” with exactly one interval in the lower layer, and they overlap in such a way that the algorithm $A$ takes first the upper layer interval, whenever available. The described “paired” intervals form the only overlapping pairs of intervals in $I_m$. Additionally, on each machine of $I_m$, there will be exactly one non-paired stand-alone interval in the lower layer. We will construct $I_m$ so that all the intervals in lower layers on all the machines correspond to different jobs. Thus, selecting all lower layer intervals forms an optimal solution.

The instance $I_m$ consists of $m$ $\alpha$-jobs and many blocks of $\beta$-jobs, where each block contains $m - 1$ jobs. The blocks are indexed by sequences of numbers up to $m$, without repetition. All the machines of $I_m$ are symmetric, just the naming of jobs differs according to each machine number, thus to describe $I_m$ we only describe the placement of jobs on a machine $M_i$. On $M_i$, there is a stand alone job $\alpha_i$, in the lower layer without a pairing interval. The remaining $m - 1$ $\alpha$-jobs are placed in the upper layer, and they are paired with the $m - 1$ intervals of the block $\beta_i$. Thus, all the intervals of $\beta_i$ are on $M_i$ in the lower layer. Then, for each $j \in \{1, \ldots, m\}$ with $j \neq i$, there is a block $\beta_j$ (again, with $m - 1$ intervals) in the upper layer. Each block $\beta_j$ is paired with a block $\beta_{ij}$, that is, the intervals of $\beta_j$ are paired with the intervals of $\beta_{ij}$ in one-to-one relation in any way. Then, for each $j, j' \in \{1, \ldots, m\}$ with $j \neq j'$ and $j, j' \neq i$, there is a block $\beta_{jj'}$ in the upper layer. Each block $\beta_{jj'}$ is then paired with $\beta_{ijj'}$ that lies in the lower layer. The construction continues in similar fashion. The construction is finalized when for each $t$, a permutation of $\{1, \ldots, m\} \setminus \{i\}$, the block $\beta_t$ is put into the upper layer and paired with $\beta_{it}$.

The main ideas of the proof are as follows. The optimum solution contains exactly all the lower-layer intervals, and thus is of size $m^2 + m(m - 1) \sum_{k=1}^{m-1} (m-1)^k!$. The instance $I_m$ is symmetric, so the general black-box algorithm produces a solution of the same size for any permutation of the machines. We consider the permutation $(1, 2, \ldots, m)$, and show that on $M_1$ the algorithm schedules $m + (m - 1) \sum_{k=1}^{m-1} (m-1)^k!$ jobs, and then on $M_i$, for $i \geq 2$, it schedules $(m - 1) \sum_{k=0}^{m-i} (m-i)^k!(k + i - 1)!$ jobs. We sum this up
and obtain the bound.

Let us first summarize the intervals on the machine \( M_i \) of the instance \( I_m \). In the upper layer there are exactly the following intervals: all the \( \alpha \)-jobs with the exception of \( \alpha_i \); and a block \( \beta_t \) for each partial permutation \( t \) of \( \{1, \ldots, m\} \setminus \{i\} \). In the lower layer there is the stand alone \( \alpha_i \); then the block \( \beta_i \) which pairs with the remaining \( \alpha \)-jobs; and finally, each block \( \beta_t \) from the upper layer is paired with the block \( \beta_{it} \) in the lower layer.

Let us now analyze the size of the optimal solution for \( I_m \). Note that the construction is made in such a way that all the intervals in the lower layer of \( M_i \) have subscripts starting with \( i \) and all the intervals in the upper layer do not contain \( i \) in their subscripts. Thus, all the lower layer intervals of \( I_m \) correspond to different jobs. This, together with the property that the intervals in lower layer do not overlap among themselves, and the fact that the lower layer contains at least as many intervals as the upper layer, implies that selecting all the lower layer intervals is not only a feasible solution, but also an optimal one.

The optimal solution on machine \( M_i \) consists of: the job \( \alpha_i \), and the following blocks of size \( m - 1 \): \( \beta_i \); \( \beta_{it} \) for each partial permutation \( t \) of \( \{1, \ldots, m\} \setminus \{i\} \). This gives altogether \( 1 + (m - 1) + (m - 1) \sum_{k=1}^{m-1} \binom{m-1}{k} k! \) jobs. Therefore, the optimal solution for \( I_m \) is of size \( m^2 + m(m - 1) \sum_{k=1}^{m-1} \binom{m-1}{k} k! \).

Let us now discuss the size of the solution obtained by the general black-box algorithm. Since the machines of the instance \( I_m \) are symmetric and the contents of any machine \( M_i \) can be mapped to the contents of any other machine \( M_j \) by just renaming the jobs (blocks), for any permutation \( \pi \) of the machines, the solution \( \text{Result}_\pi \) obtained by the general black-box algorithm is of the same size. Therefore, it is enough to consider a fixed permutation, say \( \pi = (1, 2, \ldots, m) \), and analyze the size of \( \text{Result}_\pi \).

On \( M_1 \), the general black-box algorithm selects the interval of \( \alpha_1 \) and all the intervals in the upper layer. Thus, the jobs scheduled on \( M_1 \) are exactly all the \( \alpha \)-jobs and all the blocks of \( \beta_t \)-jobs such that \( t \) is a partial permutation of \( \{1, \ldots, m\} \setminus \{1\} \). Therefore, on \( M_1 \) the algorithm schedules \( m + (m - 1) \sum_{k=1}^{m-1} \binom{m-1}{k} k! \) jobs.

On \( M_2 \), the general black-box algorithm cannot select an interval that corresponds to a job already scheduled. In particular, the \( \alpha \)-jobs, as well as the block of \( \beta_2 \)-jobs that is paired with \( \alpha \)-jobs, are not available on \( M_2 \). Also, none of the blocks of \( \beta_t \), \( t \) a partial permutation of \( \{1, \ldots, m\} \setminus \{1\} \), is available on \( M_2 \). Recall that on \( M_2 \) is each \( \beta_t \) in the upper layer paired with \( \beta_{2t} \) in the lower layer. Thus, either both subscripts of the paired
blocks contain the number 1, or none of them does. Therefore, the lower layer block is available on \( M_2 \) if and only if its pairing upper layer block is available on \( M_2 \). It follows that the algorithm on \( M_2 \) schedules exactly the blocks of \( \beta_t \)-jobs such that \( t \) is a partial permutation of \( \{1, \ldots, m\} \setminus \{2\} \) that contains the number 1. Thus, on \( M_2 \) the general black-box algorithm schedules \((m - 1) \sum_{k=0}^{m-2} \binom{m-2}{k} (k+1)! \) jobs.

Using similar arguments, we obtain that the jobs scheduled by the algorithm on \( M_i \) are exactly the blocks of \( \beta_t \)-jobs such that \( t \) is a partial permutation of \( \{1, \ldots, m\} \setminus \{i\} \) that contains all the numbers 1, 2, \ldots, \( i - 1 \). Thus, on \( M_i \) the general black-box algorithm schedules \((m - 1) \sum_{k=0}^{m-i} \binom{m-i}{k} (k + i - 1)! \) jobs.

By summing over all machines, we obtain that the algorithm on \( I_m \) schedules the following number of jobs:

\[
1 + (m - 1) \sum_{i=1}^{m} \sum_{k=0}^{m-i} \binom{m-i}{k} (k + i - 1)! = 1 + (m - 1) \sum_{i=1}^{m} \sum_{k=0}^{m} \binom{m-i}{k} (k - 1)!
\]

\[
= 1 + (m - 1) \sum_{k=1}^{m} \sum_{i=1}^{k} \binom{m-i}{k-i} (k - 1)!
\]

\[
= 1 + (m - 1) \sum_{k=1}^{m} (k - 1)! \binom{m}{m-k+1} = 1 + (m - 1) \sum_{k=1}^{m} \binom{m}{k-1} (k - 1)!
\]

Therefore, the algorithm schedules on \( I_m \) only the following fraction of the optimal solution.

\[
\frac{m + (m - 1) \sum_{k=2}^{m} \frac{m!}{(m-k+1)!}}{m^2 + m(m - 1) \sum_{k=1}^{m-1} \frac{(m-1)!}{(m-k-1)!}} = \frac{1 + (m - 1)(m - 1)! \sum_{k=1}^{m-1} \frac{1}{k!}}{m + (m - 1)(m - 1)! \sum_{k=1}^{m-1} \frac{1}{(k-1)!}}
\]

\[
= \frac{1}{(m-1)(m-1)!} + \sum_{k=0}^{m-1} \frac{1}{k!} - 1 + \sum_{k=0}^{m-2} \frac{1}{k!}
\]

\[
\xrightarrow{m \to \infty} \frac{e - 1}{e}
\]

Thus, the approximation ratio of the general black-box algorithm on INTERVALSELECTION with \( m \) machines is bounded from above by this amount. \( \square \)

### 2.4 Discussion and Related Work

We presented a general concept of \( k \)-composed problems, where the instances of the participating problems “compete” over the common ground
set. We proposed a black-box approximation algorithm for the $k$-composed problems, which in particular, improves the known results for Max-R/B-SplitGraph and INTERVALSELECTION, and provides new algorithms for MAXEDGE-\(k\)-COLORING. The class of the problems we study belongs to broader classes that have been studied before. We now describe this relation and its algorithmic consequences.

### Social Welfare Maximization Problem

The social welfare maximization problem recently received a lot of attention: Given a set of items $S$ and $k$ players with monotone (i.e., $X \subseteq Y \Rightarrow f(X) \leq f(Y)$) utility functions $f_i : 2^S \rightarrow \mathbb{R}_+$, one for every player $i \in \{1, \ldots, k\}$, the goal is to find a partition $X_1 \cup \cdots \cup X_k$ of $S$ that maximizes $\sum_{i=1}^{k} f_i(X_i)$.

The problem has been studied for various classes of utility functions, e.g., for additive, submodular, XOS (a.k.a. XOR of ORs), and subadditive functions. A set function $f$ is called XOS if there is a (finite) set of additive functions $\{a_1, \ldots, a_t\}$ such that $f(X) = \max\{a_j(X) | j \in \{1, \ldots, t\}\}$ for any $X \subseteq S$. (We note that $t$ can be exponentially large in $|S|$ and $k$.)

A $k$-composed problem $P^k$ from problems $P_1, \ldots, P_k$ can be seen as a special case of welfare maximization with XOS functions. Each of the problems $P_1, P_2, \ldots, P_k$ can be translated into a player with a XOS function as follows. Let $(I_i, w)$ be the input for $P_i$, i.e., an independence system $I_i = (S_i, F_i)$ and a weight function over $S_i$. Let $F_i^{base} \subseteq F_i$ be all the sets of maximal cardinality in $F_i$. We define the XOS function $f : 2^S \rightarrow \mathbb{R}$ for $P_i$ by the set of additive functions $\{a_1, \ldots, a_t\}$, $t = |F_i^{base}|$, so that each set $E \in F_i^{base}$ corresponds to exactly one function $a_E$ defined as $a_E(X) = \sum_{x \in X \cap E} w(x)$, for any $X \subseteq S$. Clearly, for a given set $X \subseteq S$ the function $f(X) = \max\{a_j(X) | j \in \{1, \ldots, t\}\}$ then equals to $\max\{\sum_{e \in F \cap X} w(e) | F \in F_i\}$.

The utility functions are defined over the power set of the items, and thus they contain exponential amount of information, and one has to specify how this information can be accessed. This is typically done by specifying an oracle model, which describes the queries that one can ask concerning the function $f : 2^S \rightarrow \mathbb{R}$. The most common oracles are

- **Value oracle**: for a given set $X \subseteq S$, it gives the value of $f(X)$;
- **Demand oracle**: given an assignment of prices to the items $p : S \rightarrow \mathbb{R}$, it gives a set $X \subseteq S$ that maximizes $f(X) - \sum_{j \in S} p_j$. 
Chapter 2. Composed Problems

• **XOS oracle**: assuming \( f \) is a XOS function defined by additive functions \( \{a_1, \ldots, a_t\} \), the XOS oracle for a given set \( X \subseteq S \) gives the additive function \( a_j \) such that \( f(X) = a_j(X) \).

Dobzinski et al. [26, 27] study the welfare maximization with XOS functions. Using the demand model with XOS oracle they presented a randomized \((1 - 1/e)\)-approximation algorithm, (first appeared in [27]), gave a matching upper bound of \( 1 - 1/e \), and presented a simple deterministic combinatorial 1/2-approximation algorithm for this problem. Feige [32] presented another randomized \((1 - 1/e)\)-approximation algorithm for this problem based on a different randomized rounding technique that only needs the demand oracle.

**Separable Assignment Problem**

In the *separable assignment problem (SAP)*, there is a set of \( k \) bins and \( n \) items. Assigning item \( i \) to bin \( j \) gives a profit \( v_{i,j} \). For each bin \( j \), the downward-closed family \( F_j \) describes all feasible assignments of items to bin \( j \). The goal is to find disjoint sets of items \( S_1, \ldots, S_k \) such that \( S_j \in F_j \), and the sum \( \sum_{j=1}^{n} \sum_{i \in S_j} v_{i,j} \) is maximized.

A \( k \)-composed problem \( P^k \) is a special case of SAP, where the value of an item does not depend on the bin that it is assigned to, i.e., where \( v_{i,j} = v_{i,j'} \) for all \( j, j' \).

Fleischer et al. [35] design, assuming that for each single bin subproblem there is a \( \beta \)-approximation algorithm, a randomized LP-rounding based \(((1 - 1/e)\beta)\)-approximation algorithm, and a deterministic local search \((\beta^{\beta + 1} - \epsilon)\)-approximation algorithm.

Recently, Vondrák [66] developed a randomized continuous greedy algorithm. Among other problems, he also applied it to SAP and, again assuming a \( \beta \)-approximation algorithm for a single bin subproblem, he gained a randomized \((1 - e^{-\beta})\)-approximation algorithm, thus improving upon the previous results.

**Composed Knapsack** is a special case of SAP: each item has a profit and a size, and each bin has a limited capacity. Nutov et al. [56] gave a deterministic \((1 - 1/e)\)-approximation algorithm for the problem. Even though our algorithm for composed problems does not reach this ratio, we list the above problem as an interesting specific variant of composed problems that deserves further (non black-box) study.
Maximum Coverage with Group Budgets

A lot of research has been done in the area of maximum coverage problems. A problem related to ours is the maximum coverage with group budgets (MCG) (cardinality version): For a ground set $S$ and $k$ families $X_1, \ldots, X_k$ of subsets of $S$, find a $k$-tuple $(H_1, \ldots, H_k)$, $H_i \in X_i \cup \emptyset$ maximizing $| \cup_i H_i |$. Chekuri et al. [18] defined the problem (slightly differently) and gave a deterministic greedy $1/2$-approximation algorithm for it. They note that the sets $X_i$ may be given implicitly, and thus an oracle to access the input may be needed. Dobzinski et al. [27] note that MCG is a special variant of welfare maximization with XOS functions, and thus their [27] randomized $(1 - 1/e)$-approximation algorithm also works for MCG. They also show a matching upper bound of $(1 - 1/e)$. Clearly, MCG is closely related to composed problems. However, a solution of MCG may contain intersecting sets, and hence it may not be transformable into a solution for composed problems.

Composing Identical Problems

A special case of the $k$-composed problem, arising naturally in many settings, is obtained by composing identical instances of the same problem (as opposed to composing different instances of the same problem or even of different problems). Such problems are also known as the maximum coverage problems.

For all we mention the maximum $k$-forest problem, which is polynomially solvable [37]; the maximum $k$-independent set problem, which is polynomially solvable in interval graphs [68], but in chordal graphs only for a fixed $k$ [68]; and the maximum edge $k$-coloring problem, which is APX-hard already for $k = 2$ [33]. Despite the fact that each of these particular examples is composed of identical instances of a polynomially solvable problem, their complexities differ quite a bit.

Intuitively, the problems composed of identical instances are simpler, less restrictive and thus easier to treat. For example, MaxEdge-$k$-Coloring asks for a maximum $k$-colorable subset of edges of a graph requiring adjacent edges to have different colors. Feige et al. [33] showed that a straightforward greedy algorithm (which takes in each of the $k$ turns a maximum matching, colors it with a new color and removes it from the graph), gives a $\left( 1 - \left(1 - \frac{1}{k}\right)^k \right)$-approximation; and gave a $\frac{10}{13}$-approximation algo-
Figure 2.2: In the above figures the edges in $E_R$ are marked by the dotted lines and the edges in $E_B$ are marked by the full lines. The edges $(a, b)$ and $(c, d)$ form a maximum matching for the in $G_R = (V, E_R)$ in the first figure. However, if these edges are selected into $F_R$, then the maximum matching in the remaining $G_B' = (V, E_B \setminus F_R)$ is empty. This way we obtain a solution of size 2, but optimum solution contains all 4 edges. Thus, the strategy of taking maximum matchings one after another result in only a 2-approximation when applied to the graph in the first figure. Now consider the second figure and the strategy of finding subgraphs consisting of cycles and then coloring the edges to obtains the edge $t$-coloring. Clearly, the graph $G_R$ forms a cycle. But if only the edges of this cycle are considered and colored, the solution of size 3 is obtained, whereas optimum solution contains 6 edges (the three edges of $G_B$ and some 3 edges of $G_R$). Therefore, also this strategy can be as bad as 2-approximation, when applied to the red-blue matching problem.

For the case of $k = 2$, Feige et al. present a better $\frac{10}{13}$-approximation algorithm. However, combining different instances of the same problem makes a big difference. Consider the related 2-composed problem Max-R/B-Matching (defined in Section 2.2): the deterministic strategies used for MaxEdge-$k$-Coloring give only a $1/2$-approximation (see Figure 2.2).

Concluding Remarks

As we have seen, most of the recent work on the related problems (e.g., SAP or welfare maximization with XOS functions) designs approximation algorithms based on randomized rounding methods of the corresponding LP-relaxations. Restricting ourselves to deterministic combinatorial algorithms, mainly only $1/2$-approximation algorithms are known. We believe that deterministic and purely combinatorial approaches for these problems deserve a further study.
Chapter 3

Scheduling Transfers of Resources

In car sharing services, a company manages a fleet of cars that are offered to customers for rent for a short period of time. Every car is stationed at a fixed parking location, and a customer who wishes to rent the car is usually required to return the car back to the very same location. This is a constraint that many customers would like to soften. It is thus a natural question to find alternatives allowing the customers a flexible drop-off possibility. We investigate this idea of flexible drop-offs in the case where the demands for driving (pick-up at location $A$ at time $t_A$ and drop-off at location $B$ at time $t_B$) are known in advance, and we study the problem of finding a maximum number of demands that can be realized by the existing fleet of cars and parking locations.

In this chapter we show that the problem can be solved in polynomial time by a reduction to the minimum-cost maximum-flow problem in a dedicated auxiliary graph. We further consider the problem when every user (customer) has multiple driving demands. A user is satisfied if all her demands are fulfilled (the user needs to get a car for all the requested drivings, and has no interest in partial rentals). We show that satisfying the maximum number of users is an APX-hard problem already when there are only two locations, every user has two demands, the time for driving is the same for every demand, and there is only one car. An exemplary problem that falls into this setting is the situation where a single car is used to commute between two popular, but (by public transportation) badly
Chapter 3. Scheduling Transfers of Resources

connected locations. The users want to use this car for their daily travel: Every user wants to get from one location to the other one, and later in the day also from the other location back to her original one. Interestingly, the hardness holds only whenever the travelling takes non-zero time, as we also show that for an instantaneous travel (that takes zero time), the problem becomes solvable in polynomial time.

The results in this chapter are a joint work with Yann Disser, Matúš Mihalák, and Rastislav Šrámek. They were published in LATIN 2016 [10].

3.1 Formal Problem Description and Outline

We define formally only the setting with two locations, as this setting forms the base of our main results. The problem definition for more locations is straightforward. User that rents a car at location A effectively blocks the car for a fixed time interval, and makes it available at the drop-off location B. The usage and trajectory of the car in the rental period is irrelevant for our scheduling problem, and we can simply model the renting as a transfer of the car from location A to location B at the given time interval. We abstract from our car-sharing motivation, and refer to the cars as resources.

We consider two locations, A and B, with an initial distribution of indistinguishable resources within these two locations, say there are a resources at location A and b resources at location B in the beginning. A transfer from A to B is a movement of a resource from A to B. A transfer is possible only if there is an available resource. There are n users and each of them has one or more demands: A demand d is specified by a direction X → Y (either A → B or B → A) and time interval (tsd, ted), and represents a request for a resource transfer from location X to location Y, leaving the origin X at time tsd and arriving at the destination Y at time ted. The demand d is fulfilled by moving one resource from X to Y. In this case, the resource is blocked (i.e., cannot be transferred further) for the time period (tgs, tgd), and at time tgs the number of resources at X decreases by 1 and at time tgd the number of resources at Y increases by 1. The goal is to select a feasible set of demands that maximizes the number of satisfied users. Here, a set of demands is feasible if: (i) whenever a demand of a user is selected, then all demands of the user are selected, and (ii) all selected demands can be fulfilled, i.e., we can move the resources as suggested by the demands.

We first considered the simplest questions: (1) Decide whether all users can be satisfied (or equivalently, decide whether all the demands can be
fulfilled); (2) Compute the minimum number of resources initially needed at each location to satisfy all the users. We observed that straightforward “simulation-like” algorithms can answer these questions for any number of users, demands per user, locations, and resources.

**Observation 3.1.** For any number of demands of users, for any number of locations, for any number of resources and their initial location, the problem of deciding whether all users can be satisfied is solvable in polynomial time.

*Proof.* Observe that satisfying all users is equivalent to fulfilling all demands. To check whether all demands can be fulfilled, we scan all the pick-up and drop-off events in the increasing order of time, and update the state of the system described by the number of available resources at every location. If the event is a pick-up at location \( X \), then we decrease the number of resources at \( X \) by 1. Similarly, if the event is a drop-off at location \( X \), we increase the number of resources at \( X \) by 1. Obviously, the demands can be fulfilled if and only if at every pick-up event there is at least one resource at the requested location. \( \square \)

**Observation 3.2.** For any number of demands of the users, for any number of locations, there is a polynomial-time algorithm that computes the minimum number of resources that are needed to satisfy all users.

*Proof.* Using the ideas of the previous proof, we start with zero resources at every location, and for every pick-up event that would result in a negative number of resources at the location, we simply increase the initial number of resources at that location by one. \( \square \)

In Section 3.3, we study the problem where each user has only one demand. We show that the problem of maximizing the number of satisfied users for given number of resources at locations (i.e., in this case, the number of fulfilled demands) is polynomially solvable, by reducing it to the minimum-cost maximum-flow problem. This approach works even if there are multiple locations and multiple resources in the system.

In Section 3.4, we study the variant where every user has exactly two demands: One transfer from \( A \) to \( B \) and one transfer from \( B \) to \( A \), but not necessarily in this order. Recall that a user is satisfied only if both the demands are fulfilled. We show that in this setting, it is APX-hard to maximize the number of satisfied users even if (i) there is only one resource in the system, initially placed at location \( A \), and (ii) all transfers take the same non-zero time (independently of the user and the direction). On the other hand, if the transfer time is always 0 (i.e., \( t^s_d = t^e_d \) for every
demand \( d \), we show that this problem is polynomially solvable even if there are many resources in the system.

### 3.2 Related Work

Our problem again lies in the area of interval scheduling (for recent surveys see [48, 51]), where, in the simplest case, one asks for a maximum non-intersecting subset of a given set of intervals. This simplest case would correspond to our setting with only one location \( A \) and every request of type “pick-up at \( A \) and drop-off at \( A \”).

In our problem with several locations, the transfers of a resource correspond to non-intersecting intervals (demands), with the following additional requirement: we label the interval with the corresponding pick-up and drop-off locations, and any two consecutive intervals for the same resource need to be compatible, i.e., the drop-off location of the first interval needs to be identical to the pick-up location of the second interval. For the setting with one resource and one demand per user, we ask for a maximum set of non-intersecting intervals with exactly this compatibility condition. With \( k \) resources (and one demand per user), we ask for \( k \) “chains” of such compatible solutions that together contain the maximum number of intervals (demands).

If every user has two or more demands, our problem relates to results on split intervals. A \( t \)-split interval is simply a union of \( t \) disjoint intervals. A \( t \)-interval graph is a conflict graph of \( n \) \( t \)-split intervals. Bar-Yehuda et al. [6] study the problem of finding the maximum number of non-intersecting \( t \)-split intervals, show that it is APX-hard even when \( t = 2 \), and present a 2\( t \)-approximation algorithm. Neither the approximation algorithm (or its techniques) nor the hardness result carries over to our problem with one resource and two locations. The main reason is that in our problem we require neighboring intervals in the solution to be compatible. These local compatibility requirements that we impose on the solution is what also makes our problem hard. As we will see, the hardness of our problem arises even in some configurations of intervals that would be trivially polynomially solvable under the split intervals setting (no local compatibility requirement). In particular, if every split interval intersects at most one other split interval, then the conflict graph forms a matching, and finding a maximum independent set becomes trivial. In our hardness result, in the reduction we use we obtain exactly such instances. The hardness arises
due to the compatibility requirements.

Finding the maximum number of non-intersecting split intervals with certain additional pattern requirement has been studied before with the relation to problems in RNA secondary structure prediction. The 2-interval pattern problem (see e.g., [8, 24]) asks for a non-intersecting subset of 2-split intervals such that every pair of selected split intervals is in one of the prescribed relations \( R \subseteq \{<, \sqsubset, \nmid\} \) (with \(<\) meaning preceding, \(\sqsubset\) nested, and \(\nmid\) crossed split intervals). The complexity as well as (approximation) algorithms for different subsets of \{<, \sqsubset, \nmid\} were studied.

### 3.3 Resource Transfers with One Demand per User

If every user has only one demand (either of the form \( A \to B \) or \( B \to A \)), and there are, initially, \( a \) resources at location \( A \) and \( b \) resources at location \( B \), we show that \textsc{TransfersOneDemand}, the problem of maximizing the number of satisfied users (which is in this case equal to the number of fulfilled demands), is solvable in polynomial time.

**Theorem 3.3.** \textsc{TransfersOneDemand} is solvable in polynomial time, for any number of locations.

**Proof.** We formulate the problem as a minimum-cost maximum-flow problem, which is polynomial-time solvable. For simplicity, we present this reduction considering two locations (\( A \) and \( B \)) only. The generalization for arbitrary number of locations is straightforward.

Consider an arbitrary instance of \textsc{TransfersOneDemand} with two locations. We construct an instance of the network flow problem, where the only arcs of non-zero cost correspond to the demands of the users, and have cost -1. Formally, we proceed as follows (see Figure 3.1 for illustration). For every demand \( d = (t^s_d, t^a_d) \) there are two vertices in the network, \( v^s_d \) and \( v^a_d \), one for each endpoint of \( d \). The network contains two additional vertices – a source \( s \), and a target \( t \). Based on the location of each demand’s endpoint, the corresponding vertex is either of type \( A \), or \( B \). Let \( \langle v_{a_1}, \ldots, v_{a_n} \rangle \) be the vertices of type \( A \) ordered by the time of the corresponding demands’ endpoints, and let \( \langle v_{b_1}, \ldots, v_{b_n} \rangle \) be the vertices of type \( B \) ordered by the time of the corresponding demands’ endpoints. The network contains edges of three types. For every demand \( d = (t^s_d, t^a_d) \),
there is a directed demand edge \((v^s_d, v^e_d)\) from the vertex corresponding to the startpoint of \(d\) to the vertex corresponding to its endpoint. All demand edges have capacity 1 and cost \(-1\). For every two consecutive vertices \(va_i, va_{i+1}\) of type \(A\), there is a directed connecting edge \((va_i, va_{i+1})\). Similarly, there is a connecting edge for every two consecutive vertices of \(B\). There are also connecting edges \((va_n, t)\) and \((vb_n, t)\), from the last vertices of each type to the target vertex. For all the connecting edges, the capacity is set to \(\infty\) and the cost is set to 0. Finally, there is an edge from \(s\) to vertex \(va_1\) with capacity \(a\) and cost 0, and there is an edge from \(s\) to \(vb_1\) with capacity \(b\) and cost 0. Observe that from any vertex other than \(s\), there is unlimited capacity for a flow to \(t\), using the connecting edges. Clearly, any \(st\)-flow has to pass via \(va_1\) or \(vb_1\), and the sum of capacities of \((s, va_1)\) and \((s, vb_1)\) is \(a + b\). Thus, the maximum \(st\)-flow is of size \(a + b\).

We now determine the cost of an optimum minimum-cost maximum \(st\)-flow. Since all the costs and capacities are integral, then, thanks to the integrality theorem [20], there exists an integral optimum solution, which can be found in polynomial time. From the above it follows that there is a maximum \(st\)-flow of cost 0 that does not use any demand edge. Since the network is acyclic, the capacity of a demand edge is 1, and its cost is \(-1\), it follows that a minimum-cost \(st\)-flow aims at using as many demand edges
as possible (with unit flow on each edge). We can see the integral flow as the course of the \(a+b\) resources between the locations – one \(st\)-flow of size one per resource. Obviously, an integral \(st\)-flow of cost \(-C\) satisfies \(C\) demands (users), and every schedule satisfying \(C\) users gives an integral flow of cost \(C\).

### 3.4 Resource Transfers with Two Demands per User

If the users have more than one demand, the problem of maximizing the number of satisfied users becomes NP-hard, even APX-hard. We will show the hardness even for the special case of “using the car to commute between two badly connected locations”, i.e., for the setting with two locations \(A\) and \(B\), where every user has exactly two transfer demands, one per each direction \(A \rightarrow B\), \(B \rightarrow A\). We refer to this optimization problem as \textsc{TransfersForCommuting}. We actually show that the problem is APX-hard even if there is only a single resource. We prove the hardness of \textsc{TransfersForCommuting} by an L-reduction (see Proposition 3.7) from a \((\leq 3,3)\)-\textsc{MaxSat}, which is an APX-hard variant [2] of the maximum satisfiability problem with at most 3 literals per clause and with each variable appearing in the formula at most twice as a positive, and (exactly) once as a negative literal.

**Theorem 3.4.** \textsc{TransfersForCommuting} is APX-hard even if there is only one resource, originally placed at location \(A\), and all the transfer times are equal, but positive.

First we describe a construction used to prove Theorem 3.4 and its properties.

**Construction.** Let \(\Phi\) be an instance of \((\leq 3,3)\)-\textsc{MaxSat} given by a set of clauses \(C = \{c_1, c_2, \ldots, c_r\}\) over a set \(X = \{x_1, x_2, \ldots, x_s\}\) of Boolean variables. We construct from \(\Phi\) the following instance \(I\) of \textsc{TransfersForCommuting}. There is a single resource in the system, initially located at \(A\). There are two users for each occurrence of a variable in a clause and there are 26 users for each of the \(r\) clauses, in total there are at most \(32r\) users. In the following we describe how the demands of the users are organized into gadgets, how they are placed, and how they interact.

For every variable \(x_i \in X\) there is a \textit{variable gadget} \(Gx_i\). For each clause \(c_j \in C\), there is a \textit{clause gadget} \(Gc_j\), a \textit{dummy gadget} \(DGc_j\), and a \textit{light forcing}
Chapter 3. Scheduling Transfers of Resources

Figure 3.2: Placement of the gadgets for the given instance $\Phi$ of $(\leq 3,3)$-MaxSAT with clauses $C = \{c_1, c_2, \ldots, c_r\}$ over a set of Boolean variables $X = \{x_1, x_2, \ldots, x_s\}$.

gadget $FG_{c_j}$. Finally, there are two heavy forcing gadgets $HFG_1$ and $HFG_2$. The gadgets are placed as follows (see Figure 3.2). First all the variable gadgets are placed (one per variable in $\Phi$). After that, the heavy forcing gadget $HFG_1$ is placed. Then, all the clause gadgets are placed (one per clause in $\Phi$), each preceded by a light forcing gadget. Then the heavy forcing gadget $HFG_2$ is placed. Finally, all the dummy gadgets are placed (again, one per clause in $\Phi$).

For each occurrence of a variable $x_i$ in a clause $c_j$ there is one variable user, demanding a transfer $B \rightarrow A$ first and $A \rightarrow B$ later, and a dummy variable user demanding a transfer $A \rightarrow B$ first and $B \rightarrow A$ later. Their outbound demands are placed in $G_{x_i}$. The return demand of the variable user is placed in $G_{c_j}$, and the return demand of the dummy variable user is placed in $DG_{c_j}$. For each clause $c_j$, there is one clause user, demanding an outbound transfer $B \rightarrow A$, placed in $G_{c_j}$, and a return transfer $A \rightarrow B$, placed in $DG_{c_j}$. Finally, there is a large number of forcing users, each demanding a transfer $A \rightarrow B$ and an immediate return $B \rightarrow A$, both placed in one of the forcing gadgets $FG_{c_j}$, $HFG_1$, or $HFG_2$.

We now describe the placement of the transfers within each gadget in more detail, see Figure 3.3 for the exact configurations. Each light forcing gadget $FG_{c_j}$ consists of five light forcing users, each demanding a transfer $A \rightarrow B$ and an immediate return $B \rightarrow A$. These demands are placed in such a way that all the users of a light forcing gadget can be satisfied together. Both heavy forcing gadgets $HFG_1$ and $HFG_2$ are similar to light forcing gadgets, but instead of 5 users, each HFG consists of $10r$ heavy forcing users (again demanding transfer $A \rightarrow B$ and an immediate return $B \rightarrow A$, placed in such a way that all can be satisfied together). The purpose of the light/heavy forcing gadgets is to ensure that at a certain moment the resource is located at $A$. Each forcing gadget consists of a significant number of users, such that any schedule can be transformed into the same or a larger schedule, with all forcing users satisfied.

For a variable $x_i$, the variable gadget $G_{x_i}$ consists of the outbound demands of up to six users (two for each occurrence of $x_i$ in $\Phi$). We describe the case when $x_i$ appears three times in $\Phi$, other cases are similar. The gadget $G_{x_i}$ contains 3 variable users—two positive users $x_{i,1}, x_{i,2}$ corresponding to
Figure 3.3: Placement of the demanded transfers within the gadgets (building blocks of the hardness construction). The demands of variable users are displayed as full arrows, the demands of clause users are dashed, those of dummy variable users are dotted, and those of forcing users are dash-dotted. The heavy forcing gadget is not illustrated in the figure, since it is similar to the light forcing gadget, but consists of 10r users instead of 5.

the positive literals of \( x_i \), and one negative user \( x_i \, \overline{1} \) corresponding to the negative literal of \( x_i \). Each of these users demands in \( Gx_i \) an outbound transfer \( B \to A \). Additionally, the gadget contains 3 dummy variable users \( dx_{i,1}, dx_{i,2}, \) and \( dx_{i,1} \) (complementing the variable users). Again, only their outbound demands, in direction \( A \to B \), are part of \( Gx_i \). The construction of \( Gx_i \) ensures that positive and negative variable users can never be satisfied together (the demand \( x_i \, \overline{1} \) can only be fulfilled when \( x_{i,1} \) and \( x_{i,2} \) are not, and vice versa). Moreover, the demands of each variable user and the demands of the corresponding dummy variable user can always be fulfilled together. These gadgets relate satisfying of positive/negative variable users of \( I \) with the true/false assignment of the corresponding variables in \( \Phi \).
Chapter 3. Scheduling Transfers of Resources

For a clause $c_j$, the clause gadget $G_{c_j}$ contains the return demands of up to 3 variable users (in the direction $A \rightarrow B$) that correspond to the variables appearing in $c_j$, and then an outbound demand $B \rightarrow A$ of a clause user $c_j$. (To simplify the notation we use $c_j$ to denote both the clause of $\Phi$ and the corresponding user.) Each $G_{c_j}$ is preceded by $FG_{c_j}$ enforcing that whenever the demand of $c_j$ is fulfilled in $G_{c_j}$, also a variable demand corresponding to a literal of $c_j$ is fulfilled there, and vice versa. Thus, these gadgets bind together clause and variable users: User $c_j$ is satisfied if and only if the variable user of a variable satisfying $c_j$ in $\Phi$ is satisfied.

A dummy gadget $DG_{c_j}$ consists of the return demand of the clause user $c_j$, in the direction $A \rightarrow B$, and the return demands of up to 3 dummy variable users (again, based on the literals appearing in $c_j$) in the direction $B \rightarrow A$. Gadget $DG_{c_j}$ in a sense mirrors $G_{c_j}$ and allows fulfilling the return demand of $c_j$ whenever its outbound demand is fulfilled. In a schedule where all the forcing users are satisfied, for every satisfied $c_j$, also a variable user and a dummy variable user (both corresponding to the same literal of $c_j$) will be satisfied.

Let $I$ be an instance of TransfersForCommuting constructed as above.

**Lemma 3.5.** Given a schedule $S$ of $I$, we can construct a schedule of size at least $|S|$ where all the users of heavy forcing gadgets $HFG_1$ and $HFG_2$ are satisfied.

**Proof.** If $|S| < 25r$, we can construct a schedule where all the $25r$ users of heavy and light forcing gadgets are satisfied. Thus, assume that $|S| \geq 25r$. Since the total number of users of $I$ is at most $32r$ and each HFG consists of $10r$ users, at least one user of each heavy forcing gadget HFG is satisfied. Clearly, whenever a user of a HFG is satisfied in $S$, all users of that HFG can be added to $S$. \qed

**Lemma 3.6.** Given a schedule $S$ of $I$, we can construct a schedule of size at least $|S|$ where all the users of light forcing gadgets are satisfied, and whenever a clause user $c_j$ is satisfied, also a variable user corresponding to an occurrence of a literal in $c_j$ is satisfied, as well as the corresponding dummy variable user.

**Proof.** Assume that all the heavy forcing users are satisfied in $S$ (otherwise construct such schedule using Lemma 3.5). This implies that after all the variable gadgets, the resource has to be located at $A$, and similarly, after all the clause gadgets, the resource is again located at $A$. Thus, in variable gadgets, for every fulfilled demand of a dummy variable user in the direction $A \rightarrow B$, there has to be a fulfilled demand of a variable user in the direction $B \rightarrow A$. This implies that the number of satisfied variable
users is equal to the number of satisfied dummy variable users. Similarly, in clause gadgets, the numbers of fulfilled demands of variable users and clause users coincide. This gives that the number of satisfied variable users is equal also to the number of satisfied clause users.

Due to this equivalence in numbers of satisfied variable, dummy variable, and clause users, we can argue that there exists a solution $S'$ of the same size as $S$ that exhibits a certain symmetry. In particular, we can construct it as follows. In the solution $S'$, the satisfied clause users and variable users are exactly the same as in $S$, but we remove all the dummy variable users satisfied in $S$ and replace them so that whenever a variable user is satisfied, also the corresponding dummy variable user is satisfied. Note that this is possible, since whenever a demand of a variable user is fulfilled in a variable gadget, also the demand of the corresponding dummy variable user can be fulfilled. Thus, in $S'$, the satisfied users of dummy gadgets “mirror” the satisfied users of clause gadgets. Due to this symmetry, we can divide the satisfied variable, dummy variable, and clause users into triples $(v_i, d_{v_i}, c_i)$—each containing a variable user $v_i$, its corresponding dummy variable user $d_{v_i}$, and a clause user $c_i$ such that the demands of $c_i$ are the closest fulfilled demands to the right of the second demand of $v_i$ and to the left of the second demand of $d_{v_i}$ (that is, the user $c_i$ takes the resource “back” to $A$ after $v_i$ brings it to $B$).

Now assume that there is a light forcing gadget $FG_{c_j}$ such that none of its users is satisfied. Since the users of $FG_{c_j}$ do not overlap with other users, the only problem can be the availability of the resource. Thus, there is some triple of satisfied variable, dummy variable and clause users $(v_i, d_{v_i}, c_i)$ such that the user $v_i$ transfers to $B$ just before $FG_{c_j}$ and the user $c_i$ transfers to $A$ just after $FG_{c_j}$. Then, instead of scheduling the users of this triple, we can schedule all the users of $FG_{c_j}$, and obtain a schedule that contains two more satisfied users. Also, whenever one user of a forcing gadget is satisfied, all the users of that forcing gadget can be satisfied. By series of such transformations we obtain a valid schedule $S''$ of size at least $|S'|$ where all the users of forcing gadgets are satisfied. This implies that for every triple $(v_i, d_{v_i}, c_i)$ of variable, dummy variable, and clause users (as defined above), the variable user $v_i$ and the clause user $c_i$ belongs to the same clause gadget. However, this implies that $v_i$ corresponds to an occurrence of a literal in $c_i$, which concludes the proof.

To prove APX-hardness in Theorem 3.4, we use a technique based on the following proposition.

**Proposition 3.7** (L-reduction [57]). Consider two optimization problems $H$ and
Chapter 3. Scheduling Transfers of Resources

Let \( P \) and let \( H \) be APX-hard. Assume that for each instance \( \Phi \) of \( H \), we can construct an instance \( I \) of \( P \) in polynomial time. Also, assume that for each solution \( S \) of \( I \), we can construct a solution \( \phi \) of \( \Phi \) in polynomial time. Let \( \text{OPT}(I) \), and \( \text{OPT}(\Phi) \) denote the size of the optimum solution of \( I \), and \( \Phi \), respectively. Finally, assume that there exist positive constants \( \alpha \) and \( \beta \) (independent on \( S \)) such that

\[
\text{OPT}(I) \leq \alpha \text{OPT}(\Phi)
\]

\[
|\text{OPT}(\Phi) - |\phi|| \leq \beta |\text{OPT}(I) - |S||
\]

Then, we have an L-reduction from \( H \) to \( P \), and \( P \) is also APX-hard.

Proof of Theorem 3.4. We show the APX-hardness by an L-reduction from \((\leq 3,3)\)-MaxSat. Given an instance \( \Phi \) of \((\leq 3,3)\)-MaxSat with \( r \) clauses over \( s \) variables, construct an instance \( I \) of TransfersForCommuting as above. First we show the following: \((\Rightarrow)\) For every solution \( \phi \) of \( \Phi \) of size \(|\phi|\), we construct a solution of \( I \) of size at least \( 25r + 3|\phi| \). \((\Leftarrow)\) For every solution \( S \) of \( I \) of size \(|S|\), we construct a solution \( \phi \) of \( \Phi \) of size at least \((|S| - 25r)/3\). Thus, in particular, we get \( \text{OPT}(I) = 25r + 3 \text{OPT}(\Phi) \).

\((\Rightarrow)\) Given an assignment \( \phi \) satisfying \(|\phi|\) clauses of the given instance \( \Phi \) of the \((\leq 3,3)\)-MaxSat problem, we construct a schedule where all \( 25r \) forcing users together with exactly \( 3|\phi| \) other users are satisfied as follows. We schedule \(|\phi|\) clause users corresponding to the \(|\phi|\) satisfied clauses. We select a subset of (dummy) variable users: For each clause \( c_j \), we select exactly one literal of those that satisfy \( c_j \) in \( \phi \) and we schedule both the corresponding variable user and dummy variable user. We schedule all the users of the forcing gadgets. Let us now observe that the created schedule is feasible. Clearly, the transfers of the satisfied users do not overlap: The only overlapping transfers are those of positive/negative literals in variable gadgets and those are never satisfied together, since every variable is set either to \( \text{TRUE} \), or to \( \text{FALSE} \) in \( \phi \). We now observe that the movement of the resource induced by the selected transfers is feasible. In each variable gadget the resource is moved \( A \rightarrow B \rightarrow A \) for every picked literal: \( A \rightarrow B \) by a dummy variable user and then \( B \rightarrow A \) by the variable user. After all the variable gadgets, the resource is moved \( 10r \) times \( A \rightarrow B \rightarrow A \) by the users of the forcing gadget \( \text{HFG}_1 \). In each clause gadget \( \text{Gc}_j \), the resource is moved \( A \rightarrow B \) by a variable user and then \( B \rightarrow A \) by the clause user. Before each variable gadget, the resource is moved five times \( A \rightarrow B \rightarrow A \) by the users of the forcing gadget \( \text{FGc}_j \). After the last clause gadget, the resource is moved \( 10r \) times \( A \rightarrow B \rightarrow A \) by the users of \( \text{HFG}_2 \). Finally, in each dummy gadget the resource moves \( A \rightarrow B \rightarrow A \).
(⇐) Now assume that we have a schedule $S$ with $|S|$ satisfied users. It follows from Lemma 3.5 and Lemma 3.6 that there is a schedule $S'$ of size at least $|S|$, where all $25r$ forcing users are satisfied. Moreover, it also follows that at least $(|S| - 25r)/3$ clause users are satisfied, such that for each of them also a variable user corresponding to an occurrence of a literal in $c_j$ is satisfied, as well as the corresponding dummy variable user are satisfied. Since the variable gadgets ensure that for each variable, either the users corresponding to the positive literals can be satisfied, or only the user corresponding to the negative literal can be satisfied, we can directly construct an assignment for $\Phi$ that satisfies at least $(|S| - 25r)/3$ clauses.

To show that the above reduction is an L-reduction, we need to prove that conditions (A) and (B) of Proposition 3.7 are met. First note that $\text{OPT}(\Phi) \geq r/2$ (either all-TRUE or all-FALSE assignment satisfies at least $1/2$ of all the clauses). Recall that $\text{OPT}(I) = 25r + 3 \text{OPT}(\Phi)$. Also recall that for any solution $S$ of $I$, we can construct a solution $\phi$ of $\Phi$ of size $|\phi| \geq (|S| - 25r)/3$. Thus we get:

(A) $\text{OPT}(I) = 25r + 3 \text{OPT}(\Phi) \leq 53 \text{OPT}(\Phi),$

(B) $(|\text{OPT}(I)| - |S|) \geq 25r + 3|\text{OPT}(\Phi)| - 25r - 3|\phi| = 3(|\text{OPT}(\Phi)| - |\phi|).

It follows that the presented construction is an L-reduction from $(\leq3,3)-\text{MaxSat}$ to $\text{TransfersForCommuting}$ with $\alpha = 53$ and $\beta = 1/3$.

3.4.1 Resource Transfers with Two Demands per User and Zero Transfer Times

In the hardness result, we used “crossing” arrows (demands) to exclude scheduling both demands. This is only possible if the transfer time is non-zero. In this section we show that whenever all transfer times are zero, i.e., when they are instantaneous, $\text{TransfersForCommuting}$ becomes tractable, even if there is more than one resource (initially, $a$ resources at location $A$, and $b$ resources at location $B$). As a corollary, we obtain that $\text{TransfersForCommuting}$ with non-zero transfer times is polynomially-time solvable, whenever no two demand arrows cross.

Depending on the direction of the first demanded transfer of a user, we distinguish two types of users: an $ABA$-type demands to transfer in direction $A \rightarrow B$ first, and in direction $B \rightarrow A$ second, whereas user of type $BAB$ demands first the transfer in direction $B \rightarrow A$, and later in direction $A \rightarrow B$. We can equivalently specify the problem with zero
transfer times as follows (see an example in Figure 3.4). We represent the two demands of a user $i$ (demanding instantaneous transfers at times $t_{i,1}$ and $t_{i,2}$) by a time interval $(t_{i,1}, t_{i,2})$, with a value $v(i) := -1$ if user $i$ is of type $ABA$ and $v(i) := 1$ if $i$ is of type $BAB$. Each such time interval indicates the induced change in the number of available resources present at location $A$. That is, if user $i$ of type $ABA$ transfers a resource from $A$ to $B$ at time $t_{i,1}$ and back to $A$ at time $t_{i,2}$, it implies that during the time $(t_{i,1}, t_{i,2})$ there is one less resource item at $A$ (and one more at $B$).

Clearly, satisfying a user $i$ in the original problem corresponds to selecting the corresponding interval $i$ in the modified problem. In the original problem, there must be a resource available for each selected transfer, but since the transfers are instantaneous, we only need to ensure that, at any time, both locations have a non-negative amount of resources. In particular, at $A$ there can never be more than $a + b$ items and less than 0 items. Therefore, the original goal translates to choosing a maximum subset $I$ of the intervals (representing the users) such that at any time point $t$ we have

$$-a \leq \sum_{t \in i \in I} v(i) \leq b.$$  

In the following, we show that the problem (in the equivalent alternative
formulation) is polynomially solvable by formulating it as an integral linear program (i.e., linear program that has an optimum solution which is integral). To prove that the constructed linear program $Ax \leq b$ is integral, we will show that $A$ is totally unimodular (a matrix $A$ is totally unimodular if the determinant of every square submatrix of $A$ has value -1, 0 or 1).

**Theorem 3.8 ([60]).** Every linear program in variables $x$ with a totally unimodular constraint matrix $A$ is integral.

To show that $A$ is totally unimodular, we use the following theorem [40].

**Theorem 3.9 (Ghouila-Houri).** A matrix is totally unimodular if and only if for every subset of rows $R$, there exists a function $f : R \rightarrow \{-1, +1\}$ such that $\sum_{r \in R} f(r) \cdot r \in \{-1, 0, 1\}^n$.

**Theorem 3.10.** The problem TransfersForCommuting with zero transfer time and multiple resources is solvable in polynomial time.

**Proof.** Consider the following integer linear program formulation of the problem. Let $n^+$ be the number of positive intervals (i.e., intervals of value 1) and let $n^-$ be the number of negative intervals (then $n = n^+ + n^-$). For each interval we define one variable indicating whether this interval was chosen into the optimum solution or not. In particular, for each positive interval $i$ we define a binary variable $p_i \in \{0, 1\}$ and for each negative interval $j$ we define a binary variable $n_j \in \{0, 1\}$. The goal is to maximize

$$\sum_{i=1}^{n^+} p_i + \sum_{j=1}^{n^-} n_j$$

subject to the following constraints. We divide the time axis into $N$ segments, defined by the endpoints of the $n$ given intervals. Note that the number of resources present at $A$ may change from segment to segment, but within each segment it does not change. For each segment $s$ we have the following two constraints, based on the number of intervals that overlap $s$. We abuse the notation here, and use $p_i$ and $n_j$ both as a variable and as the corresponding interval.

$$+ \sum_{i \in [n^+], \ p_i \cap s \neq \emptyset} p_i - \sum_{j \in [n^-], \ n_j \cap s \neq \emptyset} n_j \leq b \quad (3.1)$$

$$- \sum_{i \in [n^+], \ p_i \cap s \neq \emptyset} p_i + \sum_{j \in [n^-], \ n_j \cap s \neq \emptyset} n_j \leq a \quad (3.2)$$
We now consider the linear relaxation of the ILP, i.e., we additionally have the linear constraints, for every $i \in [n^+]$ and $j \in [n^-]$,

$$0 \leq p_i, n_j \leq 1.$$  \hfill (3.3)

We can write the constraints as a linear system $Ax \leq b$ (see Figure 3.4 for an example). To show that this linear program is integral, we show that the matrix $A$ is totally unimodular. For that let us first dwell into the structure of the matrix. Matrix $A$ contains one column for each variable ($p_i$ or $n_j$) and one row for each constraint. We first discuss the first $2N$ rows corresponding to the constraints (3.1) and (3.2). For each segment $s$, the matrix $A$ contains 2 rows. If the segment $s$ coincides with an interval $p_i$, then the submatrix $A(s, p_i)$ is $(1, -1)^T$, otherwise, $A(s, p_i) = (0, 0)^T$. Similarly, if $s$ coincides with an interval $n_j$, then $A(s, n_j) = (-1, 1)^T$, otherwise, $A(s, n_j) = (0, 0)^T$. Since all $p_i$ and $n_j$ are intervals, each of them spans only consecutive segments. Thus, each column (restricted to the first $2N$ rows) contains exactly one contiguous block of non-zero entries (alternating 1s and −1s). We now look at the remaining $2n = 2(n^+ + n^-)$ rows of $A$ corresponding to the constraints (3.3). Clearly, each of these rows contains exactly one non-zero symbol per row and it is either 1 or −1.

Using Theorem 3.9, we show that $A$ is totally unimodular as follows. We first observe that if the first $2N$ rows of $A$ form a totally unimodular matrix $A'$, then the whole $A$ is totally unimodular. Each of the last $2n$ rows contains exactly one non-zero element that is either 1 or −1. Thus, for every subset $R''$ of these rows, we can easily find a function $f : R'' \rightarrow \{-1, 0, 1\}^n$ so that each component of the vector $v'' = \sum_{r \in R''} f(r) \cdot r$ is either 0, or we can choose between 1 and −1. Then, for any vector $v' = \{-1, 0, 1\}^n$ (any vector obtained from $A'$ due to Theorem 3.9) we can choose the components of $v'' = \{0, -1/1\}^n$ so that $v' + v'' = \{-1, 0, 1\}^n$. It remains to be shown that the submatrix $A'$ corresponding to the first $2N$ rows of $A$ is totally unimodular. Let $R'$ be a subset of the $2N$ rows. As a preparatory step we multiply every second row of $A'$ by −1 and obtain a matrix where each column contains a single nonzero block of either 1s (if it corresponds to $p_i$) or −1s (if it corresponds to $n_j$). We set the function $g : R' \rightarrow \{-1, +1\}$ to be alternating 1 and −1 for the $r \in R'$ ordered by row number. Since each column $c$ contains only one block of consecutive 1s or −1s, we get $\sum_{r \in R'} g(c_r) \cdot c_r \in \{-1, 1\}$. Now we can combine the function $g$ with the preparatory step and obtain $f : R' \rightarrow \{-1, +1\}$ such that $\sum_{r \in R'} f(r) \cdot r \in \{-1, 0, 1\}^n$.

Thus, the matrix $A$ is totally unimodular, the constructed linear program is integral and the considered problem is polynomially solvable. \qed
Corollary 3.11. If all the demands do not cross in their arrow representation, the problem TransfersForCommuting is solvable in polynomial time.

Proof. By shrinking the given instance so that all the intervals are of length 0, we obtain an equivalent, polynomially solvable problem. \hfill \Box

3.5 Further Notes

Longest path containing subset of prescribed pairs of vertices. By proving hardness in Theorem 3.4, we prove also the following problem to be APX-hard. Given a directed graph and a set of pairs of its vertices, the goal is to find a longest path such that for each of the given pairs it either contain both vertices or none of them. This problem is APX-hard even in directed acyclic graphs, since it can be reduced from TransfersForCommuting with 1 resource as follows (see Figure 3.5). Every demand is modeled as one vertex, every user defines one prescribed pair of vertices, and there is one directed edge for every pair of demands that can be consecutively fulfilled. The constructed graph is acyclic. Clearly, any path that uses from each pair either none or both vertices corresponds to a feasible schedule for TransfersForCommuting and the length of the path corresponds to twice the number of satisfied users.

We haven’t found this exact problem to be studied in the literature, but we link to a similar problem that received a lot of attention. Given a directed graph and a set of vertex pairs, the goal of the longest antisymmetric path problem is to find a longest path that does not simultaneously contain both vertices of any of the prescribed forbidden pairs. This problem arises in the area of automatic software testing and validation, and protein identification in bioinformatics. Gabow et al. [36] showed that deciding
whether there is an antisymmetric \( st \)-path is \( \text{NP-complete} \) even if the given directed graph is acyclic and all the in- and out-degrees are at most 2. Song et al. [63] showed that the longest antisymmetric path problem cannot be approximated within \( (n - 2)/2 \) in polynomial time unless \( \text{P}=\text{NP} \), even in directed acyclic graphs of degree at most 6.

**Any of multiple demands satisfies user.** Consider a different variant of the problem, where each user has multiple demands, but is satisfied if any of her demands is fulfilled. This problem, in general form where also a transfer from a location \( L \) back to location \( L \) is allowed, is \( \text{NP-hard} \). We consider a degenerated problem as follows. There are two locations \( A \) and \( B \) and exactly one unit of resource placed at each of them. There are \( n \) users and each user demands exactly one \( A \to A \) transfer and one \( B \to B \) transfer. Thus, each user specifies exactly one interval on \( A \) and one on \( B \), we satisfy the user by selecting either of her intervals, and the goal is to maximize the number of satisfied users. This problem is then exactly \text{IntervalSelection} with 2 machines, which is \( \text{NP-hard} \), as we showed in Section 1.4.2 (Theorem 1.18).

**Open problems.** We already know (Theorem 3.4) that the problem \text{TransfersForCommuting} where each user has exactly one demand in each of the two directions is \( \text{APX-hard} \). Thus, a natural question is to seek an approximation algorithm for the problem. However, it is not clear whether it has a decent approximation, since simple approaches fail drastically. Also, given the motivation, it would be interesting to explore the problems we studied under online setting.
Chapter 4

Sequence Hypergraphs

Consider a public transportation network, e.g. a bus network, where each bus line is specified as a fixed sequence of stops. Clearly, one can travel in the network by taking a bus and following the stops in the order fixed by the corresponding line. Note that we think of a line as a sequence of stops in one direction only, since there might be one-way streets or other obstacles that cause that the bus can travel the stops in a single direction only. Then, interesting questions arise: How can one travel from $s$ to $t$ using the minimum number of lines? How many lines must break down, so that $t$ is not reachable from $s$? Are there two ways to travel from $s$ to $t$ that both use different lines?

These kinds of questions are traditionally modeled by algorithmic graph theory, but we lacked a model that would capture all the necessary aspects of the problems formulated as above. We propose the following non-standard, but a very natural way to extend the concept of directed graphs to hypergraphs.

A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ with an ordering of the vertices of every hyperedge is called a sequence hypergraph. Formally, the sequence hypergraph $\mathcal{H}$ consists of the set of vertices $\mathcal{V} = \{v_1, v_2, \ldots, v_n\}$, and the set of (sequence) hyperedges $\mathcal{E} = \{E_1, E_2, \ldots, E_k\}$, where each hyperedge $E = (v_{i_1}, v_{i_2}, \ldots, v_{i_l})$ is defined as a sequence of vertices without repetition. We remark that this definition substantially differs from the commonly used definition of directed hypergraphs [3, 4, 38], where each directed hyperedge is a pair (From, To) of disjoint subsets of $\mathcal{V}$. We note that the order of vertices in a sequence hyperedge does not imply any order of the vertices of other
hyperedges. Furthermore, the sequence hypergraphs do not impose any global order on $V$.

There is another way to look at sequence hypergraphs coming from our motivation in transportation. For a sequence hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, we construct a directed colored multigraph $G = (V, E, c)$ as follows. The set of vertices $V$ is identical to $\mathcal{V}$, and for a hyperedge $E_i = (v_1, v_2, \ldots, v_l)$ from $\mathcal{E}$, the multigraph $G$ contains $l - 1$ edges $(v_j, v_{j+1})$ for $j = 1, \ldots, l - 1$, all colored with color $c(E_i)$, with $c(E_i) \neq c(E_{i'})$ for $i \neq i'$. Therefore, each edge of $G$ is colored by one of the $k = |E|$ different colors $C = \{c(E_1), c(E_2), \ldots, c(E_k) \mid E_i \in \mathcal{E}\}$. Clearly, the edges of each color form a directed path in $G$. We refer to $G$ as the underlying colored graph of $\mathcal{H}$.

In this chapter, we study some standard graph algorithmic problems in the setting of sequence hypergraphs. In particular, we consider the problem of finding a shortest $st$-hyperpath: an $st$-path that uses the minimum number of sequence hyperedges; finding a minimum $st$-hypercut: an $st$-cut that uses the minimum number of sequence hyperedges; or finding a maximum $st$-hyperflow: a maximum number of hyperedge-disjoint $st$-hyperpaths.

We show that the shortest $st$-hyperpath is hard to approximate in general sequence hypergraphs, but can be found in polynomial time if the given sequence hypergraph is acyclic (Section 4.2). On the other hand, we show that both maximum $st$-hyperflow and minimum $st$-hypercut are APX-hard to find even in acyclic sequence hypergraphs (Sections 4.3 and 4.4). We then consider sequence hypergraphs with sequence hyperedges of constant length (defined as the number of vertices minus one). We note that the shortest $st$-hyperpath problem remains hard to approximate even with hyperedges of length at most 5, and we show that the maximum $st$-hyperflow problem remains APX-hard even with hyperedges of length at most 3. On the other hand, we show that if all the hyperedges are of length at most 2, all 3 problems become polynomially solvable (Section 4.5). We also study the complexity in a special setting in which for each hyperedge there also is a hyperedge with the same sequence, but in the opposite direction. We show that the shortest $st$-hyperpath problem becomes polynomially solvable, but both maximum $st$-hyperflow and minimum $st$-hypercut are NP-hard to find also in this setting, and we give a 2-approximation algorithm for the minimum $st$-hypercut problem (Section 4.6). Finally, we briefly study the complexity of other algorithmic problems, namely, finding a minimum spanning tree, or connected components, in sequence hypergraphs (Section 4.7). For a summary of the results see Table 4.1. The table also shows known results for the related labeled graphs (discussed below).
Table 4.1: Summary of the complexity of some classic problems in the setting of colored (labeled) graphs and sequence hypergraphs. The last row indicates whether the sizes of the maximum $st$-flow and the minimum $st$-cut equal in the considered setting. The cells in gray show our contribution.

<table>
<thead>
<tr>
<th></th>
<th>Colored/Labeled Graphs</th>
<th>Sequence Hypergraphs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>General</td>
<td>Acyclic</td>
</tr>
<tr>
<td></td>
<td>Span 1</td>
<td>Backward</td>
</tr>
<tr>
<td>Shortest $st$-path</td>
<td>APX-hard</td>
<td>P</td>
</tr>
<tr>
<td>Minimum $st$-cut</td>
<td>APX-hard</td>
<td>P</td>
</tr>
<tr>
<td>Maximum $st$-flow</td>
<td>APX-hard</td>
<td>NP-hard</td>
</tr>
<tr>
<td>MaxFlow-MinCut Duality</td>
<td>×</td>
<td>√</td>
</tr>
</tbody>
</table>

The results in this chapter are a joint work with Jérémie Chalopin, Matúš Mihalák, Guido Proietti, and Peter Widmayer. They were published in WG 2016 [9]. The result on the APX-hardness of the shortest $st$-hyperpath problem (Theorem 4.5) was published in CSR 2016 [14].

4.1 Related Work

Recently, there has been a lot of research concerning optimization problems in (multi)graphs with colored edges, where the cost of a solution is measured by the number of colors used, e.g., one may ask for an $st$-path using the minimum number of colors. The motivation comes from applications in optical or other communication networks, where a group of links (i.e., edges) can fail simultaneously and a goal is to find resilient solutions. Similar situation may occur in economics, when certain commodities are sold (and priced) in bundles.

Formally, colored graphs or labeled graphs, are (mostly undirected) graphs where each edge has one color, and in general there is no restriction on a set of edges of the same color. Note that some of the studies consider a slightly different definition of a colored graphs, where to each edge corresponds a set of colors instead of a single color. Since the computational complexity of some problems may differ in the two models, the transformations between the two models have been investigated [22].

The minimum label path problem, which asks for an $st$-path of a minimum number of colors, is NP-hard and hard to approximate [16, 17, 21, 41, 42, 69]. The 2 label disjoint paths problem, which asks for a pair of $st$-paths such that the sets of colors appearing on the two paths are disjoint, is NP-hard [44]. The minimum label cut problem, which asks for a set of edges
Chapter 4. Sequence Hypergraphs

of minimum number of colors that forms an $st$-cut, is NP-hard and hard to approximate [21, 70]. The minimum label spanning tree problem, which asks for a spanning tree using edges of minimum number of colors, is NP-hard and hard to approximate [42, 53].

Hassin et al. [42] give a $\log(n)$-approximation algorithm for the minimum label spanning tree problem and a $\sqrt{n}$-approximation algorithm for the minimum label path problem. Zhang et al. [70] give a $\sqrt{m}$-approximation algorithm for the minimum label cut problem. Fellows et al. study the parameterized complexity of minimum label problems [34]. Coudert et al. [21, 22] consider special cases when the span is 1, i.e., each color forms a connected component; or when the graph has a star property, i.e., the edges of every color are adjacent to one vertex.

Note that since most of these results consider undirected labeled graphs, they provide almost no implications on the complexity of similar problems in the setting of sequence hypergraphs. In our setting, not only we work with directed label graphs, but we also require edges of each color to form a directed path, which implies a very specific structure that, to the best of our knowledge, has not been considered in the setting of labeled graphs.

On the other hand, we are not the first to define hypergraphs with hyperedges specified as sequences of vertices. However, we are not aware of any work that would consider and explore this type of hypergraphs from an algorithmic graph theory point of view. In fact, mostly, these hypergraphs are taken merely as a tool, convenient to capture certain relations, but they are not studied further. We shortly list a few articles where sequence hypergraphs appeared, but we do not give details, since there is very little relation to our area of study. Berry et al. [7] introduce and describe the basic architecture of a software tool for (hyper)graph drawing. Wachman et al. [67] present a kernel for learning from ordered hypergraphs, a formalization that captures relational data as used in Inductive Logic Programming. Erdős et al. [29] study Sperner-families and as an application of a derived result they study the maximum number of edges of a so called directed Sperner-hypergraph.

4.2 On the Shortest $st$-Hyperpath

In this section, we consider the shortest $st$-hyperpath problem in general sequence hypergraphs and in acyclic sequence hypergraphs.
On the Shortest st-Hyperpath

Figure 4.1: In both figures, the grey-dotted curve, and the black curve depict two sequence hyperedges. a) The length of the st-hyperpath is 2, but the number of switches is 7. b) The st-hyperpath consists of two sequence hyperedges that also form a hypercycle.

**Definition 4.1 (st-hyperpath).** Let s and t be two vertices of a sequence hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. A set of hyperedges $P \subseteq \mathcal{E}$ forms a hyperpath from s to t, if the underlying (multi)graph $G'$ of the subhypergraph $\mathcal{H}' = (\mathcal{V}, P)$ contains an st-path, and $P$ is minimal with respect to inclusion. We call such an st-path an underlying path of $P$.

The length of an st-hyperpath $P$ is defined as the number of hyperedges in $P$. The number of switches of an st-hyperpath $P$ is the minimum number of changes between the hyperedges of $P$, when following any underlying st-path of $P$.

We note that each hyperpath may have multiple underlying paths. Also note that, even though the number of switches of an st-hyperpath $P$ gives an upper bound on the length of $P$, the actual length of $P$ can be much smaller than the number of switches of $P$ (see Figure 4.1a).

**Proposition 4.2.** Given a sequence hypergraph, and two vertices s and t, an st-hyperpath minimizing the number of switches can be found in polynomial time.

Such an st-hyperpath can be found, e.g., by a modified Dijkstra algorithm (starting from s, following the outgoing sequence hyperedges and for each vertex storing the minimum number of switches necessary to reach it).

While finding an st-hyperpath minimizing the number of switches is polynomially solvable in sequence hypergraphs (Proposition 4.2), we show that finding a shortest st-hyperpath (minimizing the number of hyperedges) is hard to approximate. On the other hand, if the given sequence hypergraph is acyclic, we show that the shortest st-hyperpath problem becomes polynomially solvable.

**Definition 4.3 (acyclic sequence hypergraph).** A set of hyperedges $O \subseteq \mathcal{E}$ forms a hypercycle, if there are two vertices $a \neq b$ such that $O$ contains both a hyperpath from $a$ to $b$, and a hyperpath from $b$ to $a$. A sequence hypergraph without hypercycles is called acyclic.

Observe that an st-hyperpath may also be a hypercycle (see Figure 4.1b).
Chapter 4. Sequence Hypergraphs

Figure 4.2: Finding a shortest \( st \)-hyperpath is at least as hard as the minimum set cover problem.

**Definition 4.4** (edges of a hyperedge). Let \( E = (v_1, v_2, \ldots, v_k) \) be a hyperedge of a sequence hypergraph \( H \). We call the set of directed edges \( \{e_i = (v_i, v_{i+1}) \mid i = 1, \ldots, k - 1\} \) the edges of \( E \). The edges of \( E \) are exactly the edges of color \( c(E) \) in the underlying colored graph of \( H \). The length of a hyperedge is defined as the number of its edges.

For a fixed order \( V^O = (v_1, v_2, \ldots, v_n) \) of vertices \( V \), an edge \( e \) of a hyperedge \( E \) is called a *forward* edge with respect to \( V^O \), if its orientation agrees with the order \( V^O \). Similarly, \( e \) is a *backward* edge, if its orientation disagrees with \( V^O \).

**Theorem 4.5.** Shortest \( st \)-hyperpath in sequence hypergraphs is NP-hard to approximate within a factor of \( (1 - \epsilon) \ln n \), unless \( P = NP \). If all the hyperedges are of length at most 5, the problem remains to be APX-hard.

**Proof.** We construct an approximation preserving reduction from the set cover problem. The reduction is similar to the one presented in [69], for the minimum label path problem in colored graphs. An instance \( I = (X, S) \) of the set cover problem is given by a ground set \( X = \{x_1, \ldots, x_n\} \), and a family of its subsets \( S = \{S_1, \ldots, S_m\} \). The goal is to find a smallest subset \( S' \subseteq S \) such that the union of the sets in \( S' \) contains all elements from \( X \).

The set cover problem is known to be NP-hard to approximate within a factor of \( (1 - \epsilon) \ln n \), unless \( P = NP \) [25]. Moreover, if each subset of \( S \) is of size at most 3, the problem remains to be APX-hard [23].

From \( I \) we construct a sequence hypergraph \( H = (V, E) \) as follows (cf. Figure 4.2 along with the construction). The set of vertices \( V = \{v_0, v_1, \ldots, v_n\} \) contains one vertex \( v_i \) for each element \( x_i \) of the ground set \( X \), plus one additional vertex \( v_0 \). Let \( V^O \) be the order of vertices in \( V \) naturally defined by their indices. The set of sequence hyperedges \( E = \{E_1, \ldots, E_m\} \) contains one hyperedge for each set in \( S \). For a set \( S_i \in S \), consider
the set of vertices that correspond to the elements in \( S_i \) and order them according to \( V^O \), to obtain a sequence \( Q = (v_{i_1}, v_{i_2}, \ldots, v_{i_r}) \), where \( i_1 < i_2 < \cdots < i_r \). First, let us consider the simplest case when none of the \( v_{ij}, v_{ij+1} \) (for \( j = 1, \ldots, r - 1 \)) are consecutive in the order \( V^O \), that is, \( i_j + 1 \neq i_{j+1} \). Then the sequence of the hyperedge \( E_i \) corresponding to \( S_i \) is \((v_{i_{r-1}}, v_i, v_{i_{(r-1)-1}}, v_{i_{(r-1)}}, \ldots, v_{i_1}, v_{i_1})\) (e.g., hyperedge corresponding to \( S_2 \) in Figure 4.2). In other words, \( E_i \) consists of forward edges: one forward edge \((v_{ij-1}, v_{ij})\) for each \( v_{ij} \) in \( Q \); and backward edges that connect the forward edges in the order opposite to \( Q \). Now, if for some \( j, v_{ij}, v_{ij+1} \) are consecutive vertices with respect to \( V^O \), i.e., \( i_j + 1 = i_{j+1} \), the sequence constructed as above would contain vertices repeatedly, which is not allowed (each sequence hyperedge has to map to a path in the underlying graph).

To avoid this, we construct \( E_i \) as follows. For simplicity of the explanation, instead of describing the sequence of the hyperedge, we specify \( E_i \) by listing the edges of the hyperedge and the path to which \( E_i \) maps. The hyperedge \( E_i \) consists of the same forward edges as before: one forward edge \((v_{ij-1}, v_{ij})\) for each \( v_{ij} \) in \( Q \). Whenever two or more vertices of \( Q \) are consecutive in \( V^O \), their corresponding forward edges form a path. Clearly, the forward edges of \( E_i \) then determine a set of (non-overlapping) paths \( p_1, p_2, \ldots, p_{r'} \) (uniquely ordered according to \( V^O \)). The backward edges of \( E_i \) then connect these paths in the order opposite to \( V^O \) into a single path (which specifies \( E_i \)). In particular, the last vertex of \( p_{r'} \) connects to the first vertex of \( p_{r'-1} \), the last vertex of \( p_{r'-1} \) connects to the first vertex of \( p_{r'-2} \), \ldots, and the last vertex of \( p_2 \) connects to the first vertex of \( p_1 \).

Note that the length of each sequence hyperedge \( E_i \) is bounded by \( 2|S_i| - 1 \), where \( |S_i| \) is the size of the set \( S_i \in S \) corresponding to \( E_i \). This follows from the fact that \( E_i \) consists of \( |S_i| \) forward edges and at most \( |S_i| - 1 \) backward edges to connect the forward edges. In particular, if each subset of \( S \) is of size at most 3, all the hyperedges are of length at most 5.

We set a source vertex \( s \) to \( v_0 \), and a target vertex \( t \) to \( v_n \), and we show that a shortest \( st \)-hyperpath in \( H \) of length \( k \) provides a minimum set cover for \( I \) of the same size, and vice versa. First, notice that all the forward edges (with respect to \( V^O \)) of the hyperedges in \( E \) are of the form \((v_i, v_{i+1})\) for some \( i = 0, \ldots, n-1 \). Together with the fact that \( t \) is after \( s \) in the order \( V^O \), it follows that any path from \( s \) to \( t \) in the underlying graph of \( H \) goes via all the vertices, in the order \( V^O \). Thus, there is an underlying path \( p \) of the shortest \( st \)-hyperpath \( P \) in \( H \), such that \( p \) does not use any backward edges of the hyperedges in \( E \). Clearly, by choosing a hyperedge \( E_i \) into the \( st \)-hyperpath \( P \), one also chooses its forward edges and this way “covers”
some sections of the underlying path of \( P \). Since there is a one to one mapping between the hyperedges in \( \mathcal{E} \) and the sets in \( \mathcal{S} \), by finding an \( st \)-hyperpath \( P \) of length \( k \), one finds a solution of size \( k \) to the original set cover problem. On the other hand, each solution of size \( k \) to the original set cover problem can be mapped, using the same direct one to one mapping in the opposite direction, to an \( st \)-hyperpath of length \( k \).

Thus, the described reduction is approximation preserving. By reducing from the general set cover problem we obtain the first part of the claim, and by reducing from the set cover problem with all subsets of size at most 3 we obtain the second part of the claim.

If the given sequence hypergraph is acyclic, a shortest \( st \)-hyperpath can be found in polynomial time.

**Theorem 4.6.** The problem of finding the shortest \( st \)-hyperpath in acyclic sequence hypergraphs can be solved in polynomial time.

**Proof.** Let \( \mathcal{H} = (\mathcal{V}, \mathcal{E}) \) be an acyclic sequence hypergraph. Since \( \mathcal{H} \) is acyclic, let \( \mathcal{V}^o \) be an order of the vertices \( \mathcal{V} \) such that all the edges of each hyperedge are forward edges with respect to this order. This implies that for every \( st \)-hyperpath, there is an underlying path where all the edges of each hyperedge appear consecutively (the last edge of a hyperedge \( E \) appearing in an underlying path is reachable by \( E \) from the first appearing edge of \( E \)). Therefore, finding the shortest \( st \)-hyperpath \( P \) in \( \mathcal{H} \) is the same as finding a hyperpath minimizing the number of switches, which can be done in polynomial time by Proposition 4.2.

### 4.3 On the Maximum \( st \)-Hyperflow

We consider the problem of finding a number of hyperedge-disjoint \( st \)-hyperpaths. Capturing a similar relation as in graphs (between a set of \( k \) edge-disjoint \( st \)-paths and an \( st \)-flow of size \( k \), when all the capacities are 1), for simplicity and brevity, we refer to a set of hyperedge-disjoint \( st \)-hyperpaths as an \( st \)-hyperflow.

**Definition 4.7** (\( st \)-hyperflow). Let \( s \) and \( t \) be two vertices of a sequence hypergraph \( \mathcal{H} = (\mathcal{V}, \mathcal{E}) \). Let \( \mathcal{F} \subseteq \mathcal{2}^E \) be a set of pairwise hyperedge-disjoint \( st \)-hyperpaths \( \mathcal{F} = \{P_1, \ldots, P_k\} \). Then, \( \mathcal{F} \) is an \( st \)-hyperflow of size \( |\mathcal{F}| = k \).
We show that deciding whether a given sequence hypergraph contains an \(st\)-hyperflow of size 2 is NP-hard, and thus finding a maximum \(st\)-hyperflow is inapproximable within a factor \(2 - \epsilon\) unless P=NP. This remains true even for acyclic sequence hypergraphs with all the hyperedges of length at most 3.

**Theorem 4.8.** Given an acyclic sequence hypergraph \(H = (V, E)\) with all hyperedges of length at most 3, and two vertices \(s\) and \(t\), it is NP-complete to decide whether there are two hyperedge-disjoint \(st\)-hyperpaths.

**Proof.** We construct a reduction from the NP-complete 3-Sat problem [39]. Let \(I\) be an instance of the 3-Sat problem, given as a set of \(m\) clauses \(C = \{c_1, \ldots, c_m\}\) over a set \(X = \{x_1, \ldots, x_n\}\) of Boolean variables. Recall, the goal of the 3-Sat problem is to find an assignment to the variables of \(X\) that satisfies all clauses of \(C\).

From \(I\) we construct a sequence hypergraph \(H = (V, E)\) as follows (cf. Figure 4.3 along with the construction). The set of vertices \(V\) consists of \(2 + (m + 1) + (n + 1) + \sum_{c_i \in C} |c_i|\) vertices: a source vertex \(s\), and a target vertex \(t\); a vertex \(c_i\) for each clause \(c_i \in C\) and a dummy vertex \(c_{m+1}\); a vertex \(x_j\) for each variable \(x_j \in X\) and a dummy vertex \(x_{n+1}\); and finally a vertex \(x_jc_i\) for each pair \((x_j, c_i)\) such that \(x_j \in c_i\), and similarly, \(\overline{x_j}c_i\) for each \(\overline{x_j} \in c_i\). Let us fix an arbitrary order \(C^O\) of the clauses in \(C\). The set of hyperedges \(E\) consists of \(4 + 2n + |I|\) hyperedges: There are 2 source hyperedges \((s, c_1)\) and \((s, x_1)\), and 2 target hyperedges \((c_{m+1}, t)\) and \((x_{n+1}, t)\). There are \(2n\) auxiliary hyperedges \((x_i, x_ic_k)\) and \((x_i, \overline{x_j}c_{k'})\) for \(i = 1, \ldots, n\), where \(c_k\), or \(c_{k'}\) is always the first clause (with respect to \(C^O\))
containing $x_i$, or $\overline{x}_i$, respectively. In case there is no clause containing $x_i$ (or $\overline{x}_i$), the corresponding auxiliary hyperedge is $(x_i, x_{i+1})$. Finally, there are $|I|$ \textit{lit-in-clause hyperedges} as follows. For each appearance of a variable $x_j$ in a clause $c_i$ as a positive literal there is one lit-in-clause hyperedge $(c_i, c_{i+1}, x_j c_i, x_j c_k)$, where $c_k$ is the next clause (with respect to $C^O$) after $c_i$ where $x_j$ appears as a positive literal (in case, there is no such $c_k$, then the hyperedge ends in $x_{j+1}$ instead). Similarly, if $x_j$ is in $c_i$ as a negative literal, there is one lit-in-clause hyperedge $(c_i, c_{i+1}, x_j c_i, x_j c_{k'})$, where $c_{k'}$ is the next clause containing the negative literal $x_j$ (or it ends in $x_{j+1}$).

Clearly, each hyperedge is of length at most 3. We now observe that the constructed sequence hypergraph $H$ is acyclic. All the hyperedges of $H$ agree with the following order: the source vertex $s$; all the vertices $c_1 \in C$ ordered according to $C^O$, and the dummy vertex $c_{m+1}$; the vertex $x_1$ followed by all the vertices $x_1 c_i$ ordered according to $C^O$, and then followed by the vertices $\overline{x}_1 c_i$ again ordered according to $C^O$; the vertex $x_2$ followed by all $x_2 c_i$ and then all $\overline{x}_2 c_i$; . . . ; the vertex $x_n$ followed by all $x_n c_i$ and then all $\overline{x}_n c_i$; and finally the dummy vertex $x_{n+1}$ and the target vertex $t$.

We show that the formula $I$ is satisfiable if and only if the sequence hypergraph $H$ contains two hyperedge-disjoint $st$-hyperpaths. There are 3 possible types of $st$-paths in the underlying graph of $H$: first one leads through all the vertices $c_1, c_2, \ldots, c_{m+1}$ in this order; second one leads through all the vertices $x_1, x_2, \ldots, x_{m+1}$ in this order and between $x_j, x_{j+1}$ it goes either through all the $x_j c_*$ vertices or through all the $\overline{x}_j c_*$ vertices (may differ for different $j$); and the third possible $st$-path starts the same as the first option and ends as the second one. Based on this observation, notice that there can be at most 2 hyperedge-disjoint $st$-hyperpaths: necessarily, one of them has an underlying path of the first type, while the other one has an underlying path of the second type.

From a satisfying assignment $A$ of $I$ we can construct the two disjoint $st$-hyperpaths as follows. The underlying path of one hyperpath leads from $s$ to $t$ via the vertices $c_1, c_2, \ldots, c_{m+1}$, and to move from $c_i$ to $c_{i+1}$ it uses a lit-in-clause hyperedge that corresponds to a pair $(l, c_i)$ such that $l$ is one of the literals that satisfy the clause $c_i$ in $A$. The second hyperpath has an underlying path of the second type, it leads via $x_1, x_2, \ldots, x_{n+1}$ and from $x_j$ to $x_{j+1}$ it uses the vertices containing only the literals that are not satisfied by the assignment $A$. Thus, the second hyperpath uses only those lit-in-clause hyperedges that corresponds to pairs containing literals that are not satisfied by $A$. This implies that the two constructed $st$-hyperpaths
are hyperedge-disjoint.

Let \( P \) and \( Q \) be two hyperedge-disjoint \( st \)-hyperpaths of \( H \). Let \( P \) has an underlying path \( p \) of the first type and \( Q \) has an underlying path \( q \) of the second type. We can construct a satisfying assignment for \( I \) by setting to \text{TRUE} the opposite literals than those that occur in the vertices on \( q \). Then, the hyperpath \( P \) describes how the clauses of \( I \) are satisfied by this assignment.

\[ \square \]

4.4 On the Minimum \( st \)-Hypercut

Quite naturally, we define an \( st \)-hypercort of a sequence hypergraph \( H \) as a set \( C \) of hyperedges, whose removal from \( H \) leaves \( s \) and \( t \) disconnected.

**Definition 4.9** (\( st \)-hypercort). Let \( s \) and \( t \) be two vertices of a sequence hypergraph \( H = (V, E) \). A set of hyperedges \( X \subseteq E \) is an \( st \)-hypercort, if the subhypergraph \( H' = (V, E \setminus X) \) does not contain any hyperpath from \( s \) to \( t \). The size of an \( st \)-hypercort \( X \) is \( |X| \), that is the number of hyperedges in \( X \).

For directed (multi)graphs, the famous MaxFlow-MinCut Duality Theorem [28] states that the size of a maximum \( st \)-flow is equal to the size of a minimum \( st \)-cut. In sequence hypergraphs, this duality does not hold, even in acyclic sequence hypergraphs as Figure 4.4 shows. But, of course, the size of an \( st \)-hyperflow is a lower bound on the size of an \( st \)-hypercort. We showed the maximum \( st \)-hyperflow problem to be APX-hard even in acyclic sequence hypergraphs. It turns out that also the minimum \( st \)-hypercort problem in acyclic sequence hypergraphs is APX-hard.

**Theorem 4.10.** Minimum \( st \)-hypercort in acyclic sequence hypergraphs is hard to approximate within a factor \( 2 - \epsilon \) under UGC, or within a factor \( 7/6 - \epsilon \) unless \( P=NP \).

\[ \text{Proof.} \] We construct an approximation preserving reduction from the vertex cover problem, which has the claimed inapproximability [47]. An instance
Chapter 4. Sequence Hypergraphs

Figure 4.5: Minimums $st$-hypercut is at least as hard as minimum vertex cover.

of the vertex cover problem is an undirected graph $I = (U, F)$, with the vertex set $U = \{u_1, \ldots, u_n\}$, and the edge set $F = \{f_1, \ldots, f_m\}$. The goal is to find a smallest subset $U' \subseteq U$ such that $U$ contains at least one vertex from each edge $f \in F$.

To construct from the instance $I$ an instance $I'$ of the minimum $st$-hypercut problem in acyclic sequence hypergraphs, we fix an order of the vertices and edges of $I$ as follows. Let $U^O = (u_1, \ldots, u_n)$ be an arbitrary order on the vertices of $U$. From every edge in $F$ we create an ordered edge, where the vertices of the edge are ordered naturally according to $U^O$. Let $F'$ denote the set of the created ordered edges, and let $F^O$ be the edges $F'$ ordered lexicographically according to $U^O$.

We construct the sequence hypergraph $H = (V, E)$ of $I'$ as follows (cf. Figure 4.5 along with the construction). The set of vertices $V$ consists of $3m + 2$ vertices: a source vertex $s$, a target vertex $t$, and for each edge $f_i \in F^O$, $i = 1, \ldots, m$ there are three vertices $v(i,1), v(i,2)$, and $v(i,3)$. The set of hyperedges $E$ consists of $2m + n$ hyperedges. There are $m$ source hyperedges of the form $(s, v(i,1))$, each of them connects $s$ to one vertex $v(i,1)$. There are $m$ target hyperedges of the form $(v(i,3), t)$, each of them connects one vertex $v(i,3)$ to $t$. And finally, there are $n$ vertex hyperedges, each corresponds to one of the vertices in $U$, and is constructed as follows.

For a vertex $u \in U$ of $I$, we describe the sequence of vertices in the corresponding vertex hyperedge $E_u$ iteratively. We start with an empty sequence. We consider the edges $F^O$ in order, and for each edge $f_i \in F^O$ that contains $u$ we prolong the sequence of $E_u$ as follows. If $f_i$ contains $u$ as the first vertex, we append $v(i,1)$ and $v(i,2)$ to $E_u$. Otherwise, $f_i$ contains $u$ as the second vertex, and we append $v(i,2)$ and $v(i,3)$ to $E_u$. Now consider
the edges of the obtained hyperedge $E_u$ and let us distinguish two types. First, there are edges of the form $(v_{(i,1)}, v_{(i,2)})$ and $(v_{(i,2)}, v_{(i,3)})$ for some $i$, and second, there are edges of the form $(v_{(i,2)}, v_{(i,1)}), (v_{(i,3)}, v_{(j,2)})$, and $(v_{(i,3)}, v_{(j,1)})$ for some $i < j$. Note that, due to the fact that the edges in $E^O$ are ordered lexicographically, all the edges of the vertex hyperedge $E_u$ take one of the forms described above. Also note that, due to the direct correspondence between the vertices in $U$ and vertex hyperedges, each tuple $(v_{(i,j)}, v_{(i,j+1)})$ is part of (i.e., an edge of) exactly one of the hyperedges.

Clearly, by the construction, the constructed sequence hypergraph is acyclic, since the ordering $V^O = (s, v_{(1,1)}, v_{(1,2)}, v_{(1,3)}, \ldots, v_{(m,1)}, v_{(m,2)}, v_{(m,3)}, t)$ is a topological sorting of the underlying graph $G$.

Let us now observe that there always exists a minimum $st$-hypercut that does not contain any source or target hyperedges. Notice that in the underlying graph $G$ of $H$, the only outgoing edge from $v_{(i,1)}$ leads to $v_{(i,2)}$, for $i = 1, \ldots, m$, and similarly, the only incoming edge to $v_{(i,3)}$ comes from $v_{(i,2)}$. Since for each $i$, the tuple $(v_{(i,j)}, v_{(i,j+1)})$ is an edge of exactly one hyperedge, every source hyperedge $(s, v_{(i,1)})$ in an $st$-hypercut $C$ can be substituted for the vertex hyperedge containing the edge $(v_{(i,1)}, v_{(i,2)})$; and every target hyperedge $(v_{(i,3)}, t)$ can be substituted for the vertex hyperedge containing the edge $(v_{(i,2)}, v_{(i,3)})$, and the resulting set is an $st$-hypercut of size equal or smaller than $C$. Thus, there exists an optimal solution that contains only vertex hyperedges.

Now observe that any minimum $st$-hypercut $C$ consisting of vertex hyperedges only, must for each $i = 1, \ldots, m$ “hit” either the edge $(v_{(i,1)}, v_{(i,2)})$ or $(v_{(i,2)}, v_{(i,3)})$ (i.e., one of those two edges is an edge of some hyperedge in $C$). Otherwise, the underlying graph of $(V, E \setminus C)$ would contain an $st$-path $p_i = s, v_{(i,1)}, v_{(i,2)}, v_{(i,3)}, t$.

We show that the described construction gives us an approximation preserving reduction: for an $st$-hypercut of size $k$, we can construct a solution for the vertex cover problem of the same size, and vice versa. Let $S' \subseteq H$ be an optimal solution to the instance $I'$ that contains only vertex hyperedges. Recall that there is a direct one to one mapping between the vertex hyperedges and the vertices $U$ of the instance $I$. There is also a direct mapping between each triple $(v_{(i,1)}, v_{(i,2)}, v_{(i,3)})$ and an edge from $F$. Since the solution $S'$ hits one of the edges $(v_{(i,1)}, v_{(i,2)})$ or $(v_{(i,2)}, v_{(i,3)})$ for each $i$, we can use the mapping to construct a solution $S \subseteq U$ to the instance $I$ of the original minimum vertex cover problem, such that $|S| = |S'|$. On
the other hand, every solution to the original vertex cover problem can be mapped, using the same direct one to one mapping in the opposite direction, to an $st$-hypercut of $\mathcal{H}$ of the same size.

\[ \square \]

### 4.5 Sequence Hypergraphs with Hyperedges of Length $\leq 2$

We have seen that some of the classic, polynomially solvable problems in (directed) graphs become APX-hard in sequence hypergraphs. Note that this often remains true even if all the hyperedges are of constant length. In particular, Theorem 4.5 states that the shortest $st$-hyperpath is hard to approximate even if all the hyperedges are of length at most 5; Figure 4.3 illustrates that the duality between minimum $st$-hypercut and maximum $st$-hyperflow breaks already with a single hyperedge of length 3; and Theorem 4.8 gives that the maximum $st$-hyperflow is hard to approximate even if all hyperedges are of length at most 3.

It is an interesting question to investigate the complexity of the problems for hyperedge lengths smaller than 5 or 3. We show that, if all the hyperedges of the given sequence hypergraph are of length at most 2, the shortest $st$-hyperpath, the minimum $st$-hypercut, and the maximum $st$-hyperflow can all be found in polynomial time.

**Theorem 4.11.** The shortest $st$-hyperpath problem in sequence hypergraphs with hyperedges of length at most 2 can be solved in polynomial time.

**Proof.** Consider a shortest $st$-hyperpath $P$ in a given sequence hypergraph with hyperedges of length at most 2. Clearly, whenever both edges of a hyperedge are part of an underlying path of $P$, they must appear consecutively on it. Thus, all the edges of each hyperedge appear consecutively on any underlying path of $P$. Therefore, the length of the shortest $st$-hyperpath $P$ is again the same as the minimum number of switches of $P$, and such a shortest $st$-hyperpath can be found in polynomial time (Proposition 4.2).

**Theorem 4.12.** The maximum $st$-hyperflow problem in sequence hypergraphs with hyperedges of length at most 2 can be solved in polynomial time.

**Proof.** Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a sequence hypergraph with hyperedges of length at most 2, and let $s$ and $t$ be two of its vertices. Then, using standard graph
algorithms we can find a maximum $st$-flow $f$ in the underlying directed multigraph $G$ of $\mathcal{H}$ with edge capacities 1. Thus, the flow $f$ of size $|f|$ gives us a set of $|f|$ edge-disjoint $st$-paths $p_1, \ldots, p_{|f|}$ in $G$ (note that any directed cycles in $f$ can be easily removed).

We iteratively transform $p_1, \ldots, p_{|f|}$ into a set of $st$-paths such that all the edges of each hyperedge appear on only one of these paths. Let $E = (u, v, w)$ be a hyperedge that lies on two different paths (see Figure 4.6), i.e., $(u, v) \in p_i$ and $(v, w) \in p_j$, for some $i, j \in [|f|]$. Then, $p_i$ consists of an $su$-path, edge $(u, v)$, and a $vt$-path. Similarly, $p_j$ consists of an $sv$-path, edge $(v, w)$, and a $wt$-path. Since all these paths and edges are pairwise edge-disjoint, by setting $p_i$ to consist of the $su$-path, edge $(u, v)$, and the $wt$-path; and at the same time setting $p_j$ to consist of the $sv$-path, and the $vt$-path, we again obtain two edge-disjoint $st$-paths $p_i$ and $p_j$. However, now the hyperedge $E$ is present only on $p_i$. At the same time, since each hyperedge is of length at most 2, all the edges of a hyperedge appear on any $st$-path consecutively, and any hyperedge that was present on only one of $p_i, p_j$, is not affected by the above rerouting and still is present on one of the two paths only.

Thus, the rerouting decreased the number of hyperedges present on more paths, and after at most $|E|$ iterations of this transformation we obtain $|f|$ hyperedge-disjoint $st$-paths, which gives us an $st$-hyperflow of size $|f|$. It is easy to observe that the size of the hyperflow is bounded from above by the size of the flow in the underlying multigraph. Thus, we obtained a maximum $st$-hyperflow in $\mathcal{H}$.

\begin{theorem}
The minimum $st$-hypercut problem in sequence hypergraphs with hyperedges of length at most 2 can be solved in polynomial time.
\end{theorem}

\begin{proof}
Let $\mathcal{H} = (V, E)$ be a sequence hypergraph with hyperedges of length at most 2, and let $s$ and $t$ be two of its vertices. As in proof of Theorem 4.12, we find a maximum $st$-flow $f$ (of size $|f|$) in the underlying directed multigraph $G$ of $\mathcal{H}$ and obtain a maximum $st$-hyperflow $F$ in $\mathcal{H}$ of the same size, i.e., $|F| = |f|$. Since in directed multigraphs the size of the
minimum cut equals the size of the maximum flow [28], it follows that we can find $|F|$ edges $e_1, \ldots, e_{|F|}$ of $G$ that forms a minimum cut of $G$. Observe that each of these edges corresponds to exactly one hyperedge. Thus, we obtain a set $C$ of at most $|F|$ hyperedges that forms an $st$-hypercut. Since any $st$-hypercut is bounded from below by the size of the hyperflow, $C$ is a minimum $st$-hypercut.

Note that we proved Theorem 4.13 by first constructing an $st$-hyperflow and then finding an $st$-hypercut of the same size. Since this is always possible, it follows that the equivalent of MaxFlow-MinCut Duality Theorem holds in this setting with hyperedges of length at most 2.

### 4.6 Sequence Hypergraphs with Backward Hyperedges

We consider a special class of sequence hypergraphs where for every hyperedge, there is the exact same hyperedge, but oriented in the opposite direction.

**Definition 4.14** (backward hyperedges). Let $E = (v_1, v_2, \ldots, v_k)$ be a hyperedge of a sequence hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$. We say that $E'$ is a backward hyperedge\(^1\) of $E$, if $E' = (v_k, \ldots, v_2, v_1)$. If for every $E$ of $\mathcal{E}$, there is exactly one backward hyperedge in $\mathcal{E}$, we refer to $\mathcal{H}$ as sequence hypergraph with backward hyperedges.

Such a situation arise naturally in urban public transportation networks, for instance most of the tram lines have also a “backward” line (which has the exact same stops as the “forward” line, but goes in the opposite order). We study the complexity of shortest $st$-hyperpath, minimum $st$-hypercut, and maximum $st$-hyperflow under this setting.

We show that, in this setting, we can find a shortest $st$-hyperpath in polynomial time. On the other hand, we show that minimum $st$-hypercut and maximum $st$-hyperflow remain NP-hard, and we give a 2-approximation algorithm for the minimum $st$-hypercut. Also observe in Figure 4.7 that the equivalent of MaxFlow-MinCut Duality Theorem does not hold in sequence hypergraphs with backward hyperedges. The positive results in this section are based on existing algorithms for standard hypergraphs,\(^1\)

\(^1\)Note, if $E'$ is a backward hyperedge of $E$, also $E$ is a backward hyperedge of $E'$.\n
Figure 4.7: Sequence hypergraph with backward hyperedges with minimum $st$-hypercut of size 4, and only three hyperedge-disjoint $st$-hyperpaths. For every displayed hyperedge, there is also a backward hyperedge, which is for simplicity omitted from the figure.

the negative results are obtained by a modification of the hardness proofs in Sections 4.3 and 4.4.

\textbf{Theorem 4.15.} The shortest $st$-hyperpath problem in sequence hypergraphs with backward hyperedges is in P.

\textit{Proof.} Let $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ be a sequence hypergraph with backward hyperedges, and let $s$ and $t$ be two vertices of $\mathcal{H}$. We construct a (standard) hypergraph $\mathcal{H}^* = (\mathcal{V}^* = \mathcal{V}, \mathcal{E}^*)$ from $\mathcal{H}$ in such a way that for each sequence hyperedge $E$ of $\mathcal{E}$, $\mathcal{E}^*$ contains a (non-oriented) hyperedge $E^*$ that corresponds to the set of vertices of $E$. Note that $E$ and its backward hyperedge $E'$ consist of the same set of vertices, thus the corresponding $E^*$ and $E'^*$ are the same. A shortest $st$-hyperpath$^2$ $P^*$ in (the standard) hypergraph $\mathcal{H}^*$ can be found in polynomial time. Observe that the size of $P^*$ gives us a lower bound $|P^*|$ on the length of the shortest path in the sequence hypergraph $\mathcal{H}$.

In fact, we can construct from $P^*$ an $st$-hyperpath in $\mathcal{H}$ of size $|P^*|$ as follows. Let us fix $p^*$ to be an underlying path of $P^*$. Let $(s = v_1, v_2, \ldots, v_{|P^*| + 1} = t)$ be a sequence of vertices, subsequence of $p^*$, such that for each $i = 1, \ldots, |P^*|$, there is a hyperedge $E^*$ in $P^*$ that contains both $v_i$ and $v_{i+1}$, and $v_i$ is the first vertex of $E^*$ seen on $p^*$, and $v_{i+1}$ is the last vertex of $E^*$ seen on $p^*$. Since every hyperedge $E^*$ of $\mathcal{E}^*$ corresponds to the set of vertices of some hyperedge $E$ of $\mathcal{E}$, there is a sequence of sequence hyperedges $(E_1, E_2, \ldots, E_{|P^*|})$, $E_i \in \mathcal{E}$, such that $v_i, v_{i+1}$ are vertices in $E_i$. Since $\mathcal{H}$ is sequence hypergraph with backward hyperedges, for every hyperedge $E$ of $\mathcal{E}$ and a pair its of vertices $v_i, v_{i+1}$ of $E$, there is an $v_i v_{i+1}$-hyperpath in $\mathcal{H}$ of size 1, which consists of $E$ or its backward hyperedge $E'$. Therefore, there is an $st$-hyperpath of size $|P^*|$ in $\mathcal{H}$. \hfill $\Box$

$^2$An $st$-hyperpath $P^*$ and its underlying path are defined as in sequence hypergraphs.
Theorem 4.16. The maximum st-hyperflow problem in sequence hypergraphs with backward hyperedges is NP-hard.

Theorem 4.16. We construct a reduction from the NP-complete 3-Sat problem [39]. Let $I$ be an instance of the 3-Sat problem, given as a set of $m$ clauses $C = \{c_1, \ldots, c_m\}$ over a set $X = \{x_1, \ldots, x_n\}$ of Boolean variables.

From $I$ we construct a sequence hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ as follows (cf. Figure 4.8 along with the construction). The construction is very similar to the one in proof of Theorem 4.8, so we highlight the changes in bold. One major change is that now for every hyperedge there is a backward hyperedge. For simplicity, we divide all the sequence hyperedges into pairs of mutually backward sequence hyperedges, and we refer to one sequence hyperedge of each pair as forward hyperedge (and to the other as...
its backward hyperedge). For simplicity of the construction, we describe explicitly only the forward hyperedges, and each of them implicitly defines a backward hyperedge.

Let $|I|$ be the size of $I$ (i.e., $|I| = \sum_{c_i \in C} |c_i|$). The set of vertices $\mathcal{V}$ consists of $2 + (m + 1) + (n + 1) + 3|I|$ vertices: a source vertex $s$, and a target vertex $t$; a vertex $c_i$ for each clause $c_i \in C$ and a dummy vertex $c_{m+1}$; a vertex $x_j$ for each variable $x_j \in X$ and a dummy vertex $x_{n+1}$; and finally three vertices $x_jc_i$, $ux_jc_i$, and $vx_jc_i$ for each pair $(x_j, c_i)$ such that $x_j \in c_i$, and similarly, $\overline{x_j}c_i$, $ux_jc_i$, and $vx_jc_i$ for each $\overline{x_j} \in c_i$. Let us fix an arbitrary order $C^O$ of the clauses in $C$. The set of hyperedges $\mathcal{E}$ contains $4 + 2n + 3|I|$ forward hyperedges (plus the same amount of the corresponding backward hyperedges that we do not specify explicitly). There are $2 + |I|$ source hyperedges: $(s, c_1)$, $(s, x_1)$, for each pair $(x_j, c_i)$, $x_j \in c_i$ there is $(s, ux_jc_i)$, and for each $\overline{x_j} \in c_i$ there is $(s, u\overline{x_j}c_i)$. There are $2 + |I|$ target hyperedges $(c_{m+1}, t)$, $(x_{n+1}, t)$, for each pair $(x_j, c_i)$, $x_j \in c_i$ there is $(vx_jc_i, t)$, and for each $\overline{x_j} \in c_i$ there is $(v\overline{x_j}c_i, t)$. There are $2n$ auxiliary hyperedges $(x_i, x_ic_k)$ and $(x_i, \overline{x_i}c_k')$ for $i = 1, \ldots, n$, where $c_k$, or $c_k'$ is always the first clause (with respect to $C^O$) containing $x_i$, or $\overline{x_i}$, respectively. In case there is no clause containing $x_i$ (or $\overline{x_i}$), the corresponding auxiliary hyperedge is $(x_i, x_{i+1})$. Finally, there are $|I|$ lit-in-clause hyperedges as follows. For each appearance of a variable $x_j$ in a clause $c_i$ as a positive literal there is one lit-in-clause hyperedge $(vx_jc_i, ux_jc_i, c_i, c_i+1, x_jc_i, x_jc_k)$, where $c_k$ is the next clause (with respect to $C^O$) after $c_i$ where $x_j$ appears as a positive literal (in case, there is no such $c_k$, then the hyperedge ends in $x_{j+1}$ instead). Similarly, if $x_j$ is in $c_i$ as a negative literal, there is one lit-in-clause hyperedge $(vx_jc_i, ux_jc_i, c_i, c_i+1, \overline{x_j}c_i, \overline{x_j}c_k)$, where $c_k$ is the next clause containing the negative literal $\overline{x_j}$ (or it ends in $x_{j+1}$).

We show that the formula $I$ is satisfiable if and only if the sequence hypergraph $\mathcal{H}$ contains $2 + |I|$ hyperedge-disjoint st-hyperpaths. Since there are exactly $2 + |I|$ source hyperedges, and no other hyperedge (including backward hyperedges) leads from the source vertex $s$, all these source hyperedges have to be used to get $2 + |I|$ hyperedge-disjoint st-hyperpaths. Similarly, all the target hyperedges have to be used. But then, each of the $|I|$ vertices $vx_jc_i$, or $v\overline{x_j}c_i$ has to be on one of the underlying st-paths. However, $vx_jc_i$ can only be reached (unless passing via $t$) from $ux_jc_i$ using a backward hyperedge of a lit-in-clause hyperedge. Similarly, $v\overline{x_j}c_i$ can only be reached from $ux_jc_i$ using a backward hyperedge of a lit-in-clause hyperedge. This all implies that there can be $2 + |I|$ hyperedge-disjoint st-hyperpaths only if $|I|$ of them are composed in one of the two following ways: a source hyperedge $(s, ux_jc_i)$, a backward hyperedge of some
lit-in-clause hyperedge to get from \(ux_jc_i\) to \(vx_jc_i\), and a target hyperedge \((vx_jc_i, t)\); or a source hyperedge \((s, u\overline{x_jc_i})\), a lit-in-clause backward hyperedge to get from \(u\overline{x_jc_i}\) to \(v\overline{x_jc_i}\), and a target hyperedge \((v\overline{x_jc_i}, t)\). Thus, the backward hyperedges of all \(|I|\) lit-in-clause hyperedges are used and cannot appear in the remaining \(st\)-hyperpaths. Also note that all other backward hyperedges are useless to reach \(t\) from \(s\), since they lead only backwards. This implies a situation equivalent to the one in proof of Theorem 4.8. That is, the formula \(I\) is satisfiable if and only if the sequence hypergraph \(H\) contains 2 hyperedge-disjoint \(st\)-hyperpaths, when considering forward hyperedges only, without the \(|I|\) source and \(|I|\) target hyperedges already used above.

Then, there are 3 possible types of \(st\)-paths in the underlying graph of \(H\): first one leads through all the vertices \(c_1, c_2, \ldots, c_{m+1}\) in this order; second one leads through all the vertices \(x_1, x_2, \ldots, x_{m+1}\) in this order and between \(x_j, x_{j+1}\) it goes either through all the \(x_jc_\ast\) vertices or through all the \(\overline{x_jc_\ast}\) vertices; and the third possible \(st\)-path starts the same as the first option and ends as the second one. Based on this observation, notice that there can be at most 2 hyperedge-disjoint \(st\)-hyperpaths: necessarily, one of them has an underlying path of the first type, while the other one has an underlying path of the second type.

From a satisfying assignment \(A\) of \(I\) we can construct the two disjoint \(st\)-hyperpaths as follows. One hyperpath leads from \(s\) to \(t\) via the vertices \(c_1, c_2, \ldots, c_{m+1}\), and to move from \(c_i\) to \(c_{i+1}\) it uses a lit-in-clause hyperedge that corresponds to a pair \((l, c_i)\) such that \(l\) is one of the literals that satisfy the clause \(c_i\) in \(A\). The second hyperpath has an underlying path of the second type, it leads via \(x_1, x_2, \ldots, x_{n+1}\) and from \(x_j\) to \(x_{j+1}\) it uses the vertices containing only the literals that are not satisfied by the assignment \(A\). Thus, the second hyperpath uses only those lit-in-clause hyperedges that corresponds to pairs containing literals that are not satisfied by \(A\). This implies that the two constructed \(st\)-hyperpaths are hyperedge-disjoint.

Let \(P\) and \(Q\) be two hyperedge-disjoint \(st\)-hyperpaths of \(H\) (considering only the forward hyperedges, without the source hyperedges that lead to vertices \(u\ \ast\ \ast\)). Let \(P\) has an underlying path \(p\) of the first type and \(Q\) has an underlying path \(q\) of the second type. We can construct a satisfying assignment for \(I\) by setting to TRUE the opposite literals than those that occur in the vertices on \(q\). Then, the hyperpath \(P\) suggests how the clauses of \(I\) are satisfied by this assignment.

\[\square\]

\textbf{Theorem 4.17.} The minimum \(st\)-hypercut problem in sequence hypergraphs with
Theorem 4.17. We construct a reduction from the NP-hard vertex cover problem. Let $I$ be an instance of the vertex cover problem, given as an undirected graph $I = (U, F)$, with the vertex set $U = \{u_1, \ldots, u_n\}$, and the edge set $F = \{f_1, \ldots, f_m\}$.

To construct from the instance $I$ an instance $I'$ of the minimum $st$-hypercircuit problem in acyclic sequence hypergraphs, we fix an order of the vertices and edges of $I$ as follows. Let $U^O = (u_1, \ldots, u_n)$ be an arbitrary order on the vertices of $U$. From every edge in $F$ we create an ordered edge, where the vertices of the edge are ordered naturally according to $U^O$. Let $F'$ denote the set of the created ordered edges, and let $F^O$ be the edges $F'$ ordered lexicographically according to $U^O$.

We construct the sequence hypergraph $H = (V, E)$ of $I'$ as follows (cf. Figure 4.9 along with the construction). The construction is very similar to the one in proof of Theorem 4.10, so we highlight the changes in bold. One major change is that now for every hyperedge there is a backward hyperedge. For simplicity, we divide all the sequence hyperedges into pairs of mutually backward sequence hyperedges, and we refer to one
sequence hyperedge of each pair as forward hyperedge (and to the other as its backward hyperedge). For simplicity of the construction, we describe explicitly only the forward hyperedges, and each of them implicitly defines a backward hyperedge.

The set of vertices $\mathcal{V}$ consists of $3m + 2 + 2n$ vertices: a source vertex $s$, a target vertex $t$, for each edge $f_i \in F^O$, $i = 1, \ldots, m$ there are three vertices $v_{(i,1)}, v_{(i,2)},$ and $v_{(i,3)}$, and finally, for each vertex $u_i \in U^O$, $i = 1, \ldots, n$, there are two vertices $a_i$ and $b_i$. The set of hyperedges $\mathcal{E}$ contains $2m + 3n$ forward hyperedges (plus the same amount of the corresponding backward hyperedges that we do not specify explicitly). There are $m$ source$^1$ hyperedges of the form $(s, v_{(i,1)})$, each of them connects $s$ to one vertex $v_{(i,1)}$, and there are $n$ source$^2$ hyperedges of the form $(s, a_i)$, connecting $s$ to $a_i$, for $i = 1, \ldots, n$. There are $m$ target$^1$ hyperedges of the form $(v_{(i,3)}, t)$, each of them connects one vertex $v_{(i,3)}$ to $t$, and there are $n$ target$^2$ hyperedges of the form $(b_i, t)$, connecting $a_i$ to $t$, for $i = 1, \ldots, n$. And finally, there are $n$ vertex hyperedges, each corresponds to one of the vertices in $U$, and is constructed as follows.

For a vertex $u_k \in U$ of $I$, we describe the sequence of vertices in the corresponding vertex hyperedge $E_{u_k}$ iteratively. We start with a sequence containing $(b_k, a_k)$. We consider the edges $F^O$ in order, and for each edge $f_i \in F^O$ that contains $u$ we prolong the sequence of $E_{u_k}$ as follows. If $f_i$ contains $u$ as the first vertex, we append $v_{(i,1)}$ and $v_{(i,2)}$ to $E_{u_k}$. Otherwise, $f_i$ contains $u$ as the second vertex, and we append $v_{(i,2)}$ and $v_{(i,3)}$ to $E_{u_k}$. Now consider the edges of the obtained hyperedge $E_{u_k}$ and let us note what form they can have. First, there is an edge $(b_k, a_k)$, then $(a_k, v_{(i,j)})$ for some $j \in \{1, 2\}$ and some $i \in [m]$, and the remaining edges are of two main types: There are edges of the form $(v_{(i,1)}, v_{(i,2)})$ and $(v_{(i,2)}, v_{(i,3)})$ for some $i \in [m]$, and there are edges of the form $(v_{(i,2)}, v_{(j,1)})$, $(v_{(i,3)}, v_{(j,2)})$, and $(v_{(i,3)}, v_{(j,1)})$ for some $i < j$, $i, j \in [m]$. Note that, due to the fact that the edges in $F^O$ are ordered lexicographically, all the edges of the vertex hyperedge $E_{u_k}$ take one of the forms described above. Also note that, due to the direct correspondence between the vertices in $U$ and vertex hyperedges, each tuple $(v_{(i,j)}, v_{(i,j+1)})$ is part of (i.e., an edge of) exactly one of the hyperedges.

Let us now observe that there always exists a minimum $st$-hypercut that does not contain any source or target hyperedges. Clearly, none of the backward source or target hyperedges are in a minimum $st$-hypercut, since each consists of a single edge that either leads to $s$ or from $t$. Notice that in the underlying graph $G$ of $\mathcal{H}$, the only outgoing edge from $a_k$ (other than
to $s$) leads to $b_k$, for $k = 1, \ldots, n$, and similarly, the only incoming edge to $b_k$ (other than from $t$) comes from $a_k$. Since for each $k$, the tuple $(a_k, b_k)$ is an edge of exactly one hyperedge, in particular of a backward hyperedge of some vertex hyperedge, every forward source\(^2\) or target\(^2\) hyperedge in an $st$-hypercut $C$ can be substituted for the backward vertex hyperedge containing the edge $(a_k, b_k)$, and the resulting set is an $st$-hypercut of size equal or smaller than $C$. Thus, there exists a minimum $st$-hypercut $C$ which does not contain source\(^2\) or target\(^2\) hyperedges. Now observe that such an $st$-hypercut $C$ has to contain all the backward vertex hyperedges, as otherwise the underlying graph of $(\mathcal{V}, \mathcal{E} \setminus C)$ would contain an $st$-path $q_k = s, a_k, b_k, t$, for some $k$. Now we show that each source\(^1\) or target\(^1\) hyperedge in $C$ can be substituted for a forward vertex hyperedge. Let $B$ be the set of all backward vertex hyperedges. Notice that in the underlying graph of $(\mathcal{V}, \mathcal{E} \setminus B)$, the only outgoing edges from $v(i,1)$ lead to $s$ or $v(i,2)$, for $i = 1, \ldots, m$; and similarly, the only incoming edges to $v(i,3)$ come from $t$ or $v(i,2)$. (And clearly, the hyperedges $(v(i,1), s)$ and $(t, v(i,3))$ are not part of any $st$-hyperpath.) Since for each $i$, the tuple $(v(i,j), v(i,j+1))$ is an edge of exactly one hyperedge, every source\(^1\) hyperedge $(s, v(i,1))$ in an $st$-hypercut $C$ can be substituted for the forward vertex hyperedge containing the edge $(v(i,1), v(i,2))$; and every target\(^1\) hyperedge $(v(i,3), t)$ can be substituted for the forward vertex hyperedge containing the edge $(v(i,2), v(i,3))$, and the resulting set is an $st$-hypercut of size equal or smaller than $C$. Thus, there exists an optimum solution that contains only vertex hyperedges, and in particular, it contains all the backward vertex hyperedges.

Now observe that any minimum $st$-hypercut $C$ consisting of vertex hyperedges only, must for each $i = 1, \ldots, m$ “hit” either the edge $(v(i,1), v(i,2))$ or $(v(i,2), v(i,3))$ (i.e., one of those two edges is an edge of some hyperedge in $C$). Otherwise, the underlying graph of $(\mathcal{V}, \mathcal{E} \setminus C)$ would contain an $st$-path $p_i = s, v(i,1), v(i,2), v(i,3), t$.

Let $S' \subseteq \mathcal{H}$ be an optimum solution to the instance $I'$ that contains only vertex hyperedges. Since $S'$ contains all $n$ backward vertex hyperedges, it contains $|S'| - n$ forward vertex hyperedges. Recall that there is a direct one to one mapping between the forward vertex hyperedges and the vertices $U$ of the instance $I$. There is also a direct mapping between each triple $(v(i,1), v(i,2), v(i,3))$ and an edge from $F$. Since the solution $S'$ hits one of the edges $(v(i,1), v(i,2))$ or $(v(i,2), v(i,3))$ for each $i$, we can use the mapping to construct a solution $S \subseteq U$ to the instance $I$ of the original minimum vertex cover problem, such that $|S| = |S'| - n$. On the other hand, every solution of size $k$ to the original vertex cover problem can
be mapped, using the same direct one to one mapping in the opposite
direction, to an \( st \)-hypercut of \( \mathcal{H} \) of size \( k + n \). Therefore, an optimum
solution for \( I' \) gives us an optimum solution for \( I \).

**Theorem 4.18.** The minimum \( st \)-hypercut problem in sequence hypergraphs with
backward hyperedges can be 2-approximated.

*Theorem 4.18.* Let \( \mathcal{H} = (\mathcal{V}, \mathcal{E}) \) be a sequence hypergraph with backward
hyperedges, and let \( s \) and \( t \) be two vertices of \( \mathcal{H} \). Note that we can
partition \( \mathcal{E} \) into \( |\mathcal{E}|/2 \) pairs of hyperedges (each pair contains a hyperedge
and its backward hyperedge). We construct a (standard) hypergraph
\( \mathcal{H}^* = (\mathcal{V}^*, \mathcal{E}^*) \) from \( \mathcal{H} \) in such a way that for each pair of mutually
backward sequence hyperedges \( E, E' \) of \( \mathcal{E} \), \( \mathcal{E}^* \) contains exactly one (non-
oriented) hyperedge \( E^* \) that corresponds to the set of vertices of \( E \) (and
also \( E' \)). Note that \( \mathcal{E}^* \) may contain multiple hyperedges corresponding
to the same set of vertices, and we get that \( \mathcal{E}^* \) contains exactly \( |\mathcal{E}|/2 \)
hyperedges.

Next, we find a minimum \( st \)-hypercut \( X^* \) in \( \mathcal{H}^* \) (can be done in polynomial
time by a transformation of \( \mathcal{H}^* \) to a directed graph and solving a maximum
flow problem in it). Clearly, \( |X^*| \) is the lower bound on the size of a
minimum \( st \)-hypercut in \( \mathcal{H} \). Recall that every hyperedge in \( X^* \) corresponds
to 2 sequence hyperedges in \( \mathcal{H} \), thus, by removing all sequence hyperedges
corresponding to hyperedges in \( X^* \), we obtain an \( st \)-hypercut in \( \mathcal{H} \) of size
\( 2 \ |X^*| \).

4.7 On Other Algorithmic Problems

We briefly consider some other standard graph algorithmic problems in
sequence hypergraphs.

**Definition 4.19** (rooted spanning hypergraph). Let \( \mathcal{H} = (\mathcal{V}, \mathcal{E}) \) be a se-
quence hypergraph. We define \( s \)-rooted spanning hypergraph \( T \) as a subset of
\( \mathcal{E} \) such that for every \( v \in \mathcal{V} \), \( T \) is an \( sv \)-hyperpath. The size of \( T \) is defined
as \( |T| \).

**Theorem 4.20.** Minimum \( s \)-rooted spanning hypergraph in acyclic sequence
hypergraphs is NP-hard to approximate within a factor of \( (1 - \varepsilon) \ln n \), unless
\( P = NP \).
Proof. We construct an approximation preserving reduction from the set cover problem, which has the claimed inapproximability [25]. Let \( I = (X, S) \) be an instance of the set cover problem, given by a ground set \( X = \{x_1, \ldots, x_n\} \), and a family of its subsets \( S = \{S_1, \ldots, S_m\} \).

We construct from \( I \) an instance \( I' \) of the minimum \( s \)-rooted spanning hypergraph problem in sequence hypergraphs (see Figure 4.10 for an example). Let \( H = (V, E) \) be a sequence hypergraph as follows. The set of vertices \( V = \{s, v_1, \ldots, v_n\} \) contains one vertex \( v_i \) for each \( x_i \in X \), and a source vertex \( s \). Let \( V^O \) be an arbitrary fixed order on vertices \( V \). There are \( m \) hyperedges in \( E = \{E_1, \ldots, E_m\} \), one for each set in \( S \). For a set \( S_i \in S \), let us take the vertices that correspond to the elements in \( S_i \) and order them according to \( V^O \) to obtain \((v_{i_1}, v_{i_2}, \ldots, v_{i_r})\). Then the sequence of the hyperedge \( E_i \) corresponding to \( S_i \) is simply \((s, v_{i_1}, v_{i_2}, \ldots, v_{i_r})\). The constructed sequence hypergraph is acyclic, as all the edges of its hyperedges agree with \( V^O \).

It remains to show that from a minimum \( s \)-rooted spanning hypergraph \( T \) for \( I' \) can be constructed a minimum set cover for \( I \) of the same size. By definition, \( T \) is the smallest subset of \( E \) that allows to reach from \( s \) any other vertex in \( V \). Moreover, since \( H \) is acyclic and each hyperedge starts at \( s \), for each vertex \( v \in V \), it is enough to choose one of the hyperedges in \( E \) (containing \( v \)) to reach \( v \) from \( s \). This, together with the one-to-one correspondence between the hyperedges in \( E \) and sets in \( S \), provides us with a prescription for a minimum set cover for \( I \) of size \(|T|\). On the other hand, we can map any solution of the original minimum set cover to a minimum \( s \)-rooted spanning hypergraph of the same size.

\[ \square \]

**Definition 4.21** (strongly connected component). Let \( H = (V, E) \) be a sequence hypergraph. We say that a set \( C \subseteq E \) forms a strongly connected component if for every two vertices \( u, v \in V', V' \) being all the vertices of \( V \)
present in $C$, the set $C$ contains a $uv$-hyperpath. We say that the vertices in $\mathcal{V}'$ are covered by $C$.

Clearly, we can decide in polynomial time whether the given set of hyperedges $C$ forms a strongly connected component as follows. Consider the underlying graph $G$ of $\mathcal{H}$ induced by the set of sequence hyperedges $C$ and find a maximum strongly connected component there. If this component spans the whole $G$, then $C$ is a strongly connected component in $\mathcal{H}$.

**Theorem 4.22.** Given a sequence hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$, it is NP-hard to find a minimum number of hyperedges that form a strongly connected component $C$ so that a) $C$ is any non-empty set, or b) all the vertices in $\mathcal{V}$ are covered by $C$.

**Proof.** Variant b) is clearly NP-hard by a reduction from Hamiltonian cycle: Consider a standard directed graph $G = (\mathcal{V}, \mathcal{E})$ and see it as a sequence hypergraph, where each sequence hyperedge is just an edge of $G$. Then, by finding a strongly connected component that covers all the vertices of $\mathcal{V}$ and uses the minimum number of sequence hyperedge (= normal edges) we can solve the Hamiltonian cycle problem in $G$ (either the component consists of $|\mathcal{V}|$ edges or more).

We show NP-hardness of variant a) by a reduction from the NP and APX-hard set cover problem [25]. Let $I = (X, S)$ be an instance of the set cover problem, given by a ground set $X = \{x_1, \ldots, x_n\}$, and a family of its subsets $S = \{S_1, \ldots, S_m\}$.

From $I$ we construct a sequence hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ as follows (see Figure 4.11 for an example). The set of vertices $\mathcal{V} = \{s, t, v_1 \ldots, v_n, u_1 \ldots, u_n\}$ contains one vertex $v_i$ and one vertex $u_i$ for each $x_i \in X$, a source vertex $s$, and a target vertex $t$. Let $X^O$ be an arbitrary fixed order on the elements of $X$. The set of sequence hyperedges $\mathcal{E}$ consists of $m + n + 1$
hyperedges. There are $n$ element hyperedges: for each element $x_i$ in $X$ (i-th element of $X$ in the order $X^O$), there is a hyperedge $(v_i, u_i, u_{i-1})$, the hyperedge $(v_1, u_1, s)$ corresponding to $x_1$ ends at $s$ instead. There is one target hyperedge $(t, u_n)$. Finally, there are $m$ set hyperedges, one for each set in $S$ defined as follows. For a set $S_i \in S$, let us take the vertices $v_*$ that correspond to the elements in $S_i$ and order them according to $X^O$ to obtain $(v_{i_1}, v_{i_2}, \ldots, v_{i_r})$. Then the sequence of the hyperedge $E_i$ corresponding to $S_i$ is simply $(s, v_{i_1}, v_{i_2}, \ldots, v_{i_r}, t)$.

It remains to show that from a minimum non-empty set of sequence hyperedges of $\mathcal{H}$ that form a strongly connected component, can be constructed a minimum set cover for $I$. Let $O$ be such a non-empty set of hyperedges. First observe that if $O$ contains any set hyperedge, then $O$ must contain all the element hyperedges and the target hyperedge. This follows from the fact, that if $O$ contains a set hyperedge, then $s$ and $t$ are covered by $O$, and $O$ must contain a $ts$-hyperpath, but the only $ts$-hyperpath clearly consists of all element hyperedges plus the target hyperedge. Clearly, if $O$ contains an element hyperedge or a target hyperedge, then either some $v_i$ or $t$ is covered by $O$, and thus $O$ must contain a hyperpath that leads to $v_i$ or $t$, respectively. However, the only sequence hyperedges that can reach $v_i$ or $t$ are the set hyperedges. Therefore, $O$ always contains at least one set hyperedge and all the element hyperedges and the target hyperedge. However, this implies that all the vertices have to be covered by $O$, which, in particular, implies that $O$ must contain an $sv_i$-hyperpath for each $v_i$. Therefore, $O$ consists of all element hyperedges, the target hyperedge and the smallest subset of the set hyperedges that allows to reach from $s$ any vertex $v_i \in V$. In other words, for each vertex $v_i \in V$, $O$ must contain a set hyperedge that can reach $v_i$ from $s$. This, together with the one-to-one correspondence between the set hyperedges in $\mathcal{E}$ and sets in $S$, provides us with a prescription for a minimum set cover for $I$ of size $|O| - n - 1$. On the other hand, using the same mapping in the opposite direction, we can map any solution of the original minimum set cover to a minimum non-empty set of sequence hyperedges of $\mathcal{H}$ that form a strongly connected component.

**Theorem 4.23.** Given a sequence hypergraph $\mathcal{H} = (V, \mathcal{E})$, finding a maximum number of hyperedges that form a strongly connected component $C$ so that a) $C$ is any non-empty set, or b) all the vertices in $V$ are covered by $C$, is polynomial-time solvable.

**Proof.** As discussed above, for a given set of hyperedges, we can decide in polynomial time whether it forms a strongly connected component or
not. Thus, in particular, we can check whether the set \( E \) forms a strongly connected component. In case it does, variant b) is solved trivially by taking all the hyperedges in \( E \). Otherwise, this variant has no solution, because there is no strongly connected component covering all vertices in \( \mathcal{V} \).

To solve variant a), we start with a set of all hyperedges \( E \) and during the process, we iteratively remove those hyperedges that cannot be part of a feasible solution. Iteratively, we find strongly connected components in the underlying graph induced by the current set of hyperedges, and remove all the sequence hyperedges that occur in two or more components (as these cannot be a part of a single strongly connected component) and repeat. Once no more hyperedges can be removed, we reached a situation, where each of the remaining hyperedges is in exactly one strongly connected component. Thus, the solution is defined by the component that contains the maximum number of hyperedges. \( \square \)
Concluding Remarks

In this thesis, we have studied various algorithmic questions arising from passenger transportation. In the course of our research, before we dived into solving each problem at hand, we first aimed to develop a model that would capture the essence of the problem, and was at the same time simple and felt natural.

The INTERVAL SELECTION problems belong to a widely studied area of interval scheduling, where many variants continue to appear, coming from very diverse applications. We studied the computational complexity of various restricted variants, trying to gain an understanding what makes the problem hard. Apparently, the variant with cores is a particular setting that arises naturally from different applications, thus repeatedly attracts attention (one of our hardness results addresses a previously posed open question, and more recently, the complexity of even more restricted variants of this setting has been explored). Given the hardness of the problems in this setting, a natural tractable approach are approximation algorithms. Even though there is no hope for a PTAS (since the problems are APX-hard), we find it interesting to explore deterministic options offering a middle ground between the known exact algorithm which is exponential in the number of machines, and the simple greedy algorithm which gives a 1/2-approximation. As a tiny step in this direction, with the 501/1000-approximation algorithm we demonstrated that the barrier of 1/2 can be beaten even if we don’t limit the number of machines. Considering a different restricted variant of the general problem we proposed the deterministic 2/3-approximation algorithm for the case with two machines, again improving upon the previously known 1/2-approximation. An interesting task for future research is to design approximation algorithms with better approximation guarantees.
Even though the 2/3-approximation algorithm was initially designed for a very specific variant of INTERVAL SELECTION, due to the simplicity of the key idea of this algorithm it was possible to generalize it for a much broader class of problems. We are aware that there are randomized algorithms that achieve better approximation ratios, but often they are based on solving LP-relaxations and the successive randomized rounding, and this makes it difficult to see how the structure of the problem is related to the solution. We believe that the main strength of our algorithm is its simplicity, and because of that it might be possible to generalize it even more, in directions we haven’t considered, or to use it on problems that require specific properties of the solution that are hard to assure with complicated algorithms.

We have seen how the optimization problems in car-sharing with flexible drop-offs naturally translate into the general optimization problem of scheduling transfers of resources. For the primary variant with one demand for user we provided a polynomial-time solution, which can be applied to the original real-world problem. In general, the problems arising in this setting can be seen as interval scheduling problems, where we require the solution to have a certain pattern. Similar requirement arise in various applications, not only in interval scheduling, so maybe a similar model can be used also in other areas. We think that our motivation and the setting offer more problems to explore. In particular, it might be interesting to study the problem in an online setting, as well as new variants motivated by further considerations of the real-world applications (e.g., resilience to a last minute decision of a user not to take the car, usage of self-driving cars, etc.).

In the last chapter we defined sequence hypergraphs, a graph-theoretic model for transportation lines. With this model we were able to easily formulate some of the algorithmic questions motivated by concerns about network’s resilience to failures, as well as provide some answers on their computational complexity. At the same time, we feel that the model is interesting also from a graph-theory point of view, as it offers an alternative way to generalize the concept of a directed edge from graphs to hypergraphs. If all the hyperedges of a sequence hypergraph are of length 1, we get a standard directed graph. We have seen that some classic optimization problems that are polynomially solvable in directed graphs are APX-hard in sequence hypergraphs. Interestingly, we have also seen that this complexity does not come immediately with the step from directed graphs to sequence hypergraphs, since in the case of sequence hypergraphs with hyperedges of length at most 2, these problems remain solvable in polynomial time.
We believe that sequence hypergraphs deserve further study and offer plenty to explore, both from algorithmic and graph-theory perspective. Clearly, the problems we considered can be studied further, for example, designing non-trivial approximation algorithms is a largely unexplored area. We think it might be interesting to explore also other variants of these hypergraphs, to consider wider range of algorithmic problems under this setting, and to study structural properties of sequence hypergraphs.


Curriculum Vitae

Kateřina Böhmová

Birth date February 17, 1987
Birth place Czech Republic
Nationality Czech

Education

2012 – 2016 PhD in Theoretical Computer Science
ETH Zurich, Zurich

2009 – 2011 MSc in Applied Mathematics
UPC BarcelonaTech, Barcelona

2006 – 2009 BSc in Computer Science
Charles University, Prague

1998 – 2006 Maturita (high school final exam)
Gymnázium, Rožnov pod Radhoštěm
List of Publications

Conferences


Submitted

K. Böhmová, E. Kravina, and M. Mihalák. Maximization problems competing over a common ground set and their black-box approximation. Unpublished manuscript.