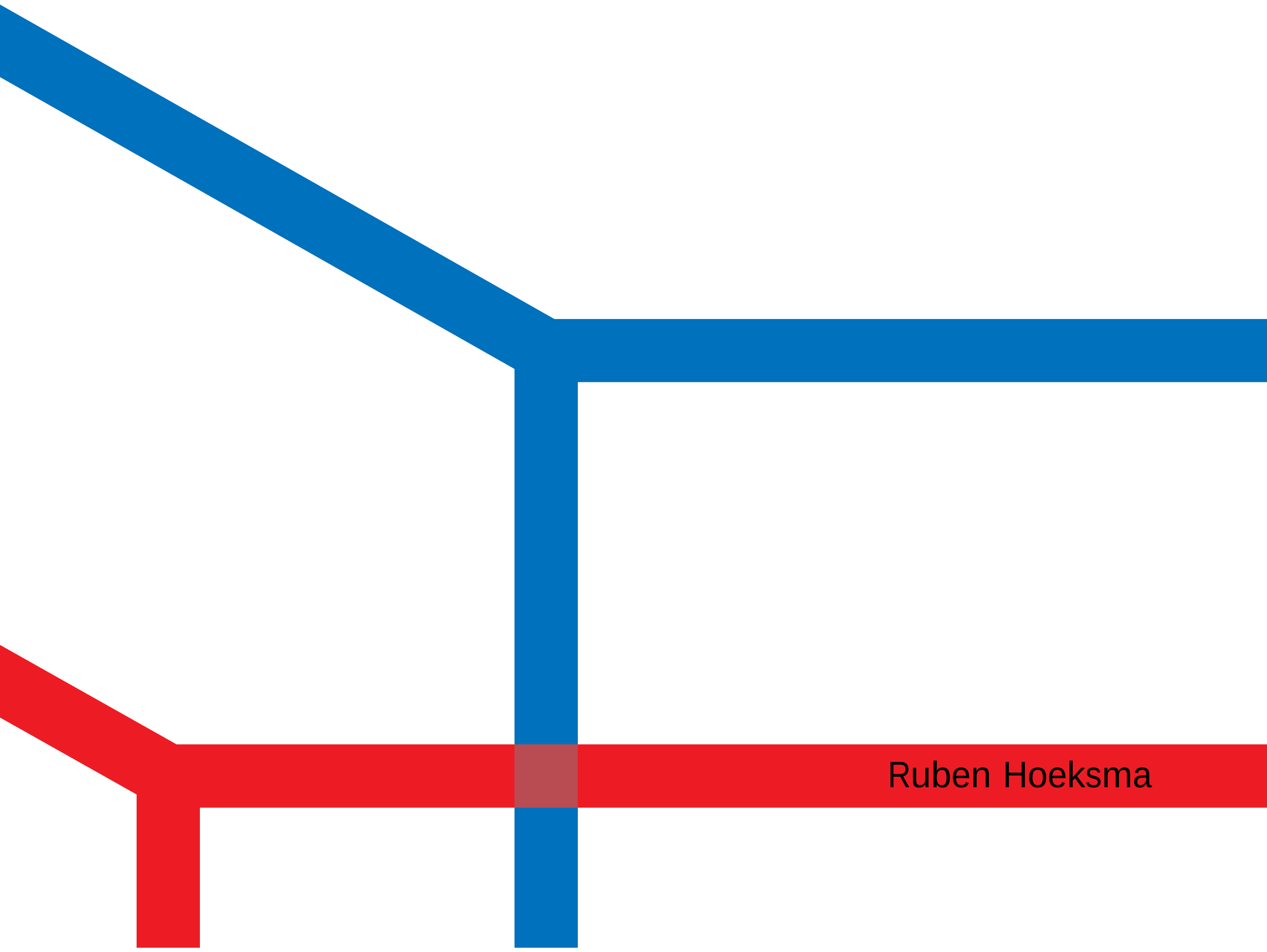
Mechanisms for scheduling games with selfish players



Ruben Hoeksma

Mechanisms for scheduling games with selfish players

Ruben Hoeksma

CTIT

CTIT Ph.D. Thesis Series No. 14-342
Centre for Telematics and Information Technology
University of Twente
P.O. Box 217, NL – 7500 AE Enschede

ISSN: 1381-3617 (CTIT Ph.D. Thesis Series No. 14-342)
ISBN 978-90-365-3827-5
DOI: 10.3990/1.9789036538275
`http://dx.doi.org/10.3990/1.9789036538275`

Typeset with LaTeX. Printed by Ipskamp Drukkers.

Cover: *Abstraction of the scheduling polytope for four jobs.*
Design by P.H. Hoeksma.

MECHANISMS FOR SCHEDULING GAMES WITH SELFISH PLAYERS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
Prof.dr. H. Brinksma,
volgens besluit van het College voor Promoties,
in het openbaar te verdedigen
op vrijdag 30 januari 2015 om 16.45 uur

door

Ruben Pieter Hoeksma

geboren op 6 maart 1985
te Nijmegen, Nederland

Dit proefschrift is goedgekeurd door:
Prof.dr. M.J. Uetz

Acknowledgments

A Ph.D. thesis is the product of several years of work and, therefore, influenced by a lot of people. Here I would like to thank all of you who contributed, in one way or another, to me being able to finish this work. The following people deserve a special mention.

First of all I thank Marc Uetz for the supervision during this trajectory. Starting with the final project of my master's degree and followed by my Ph.D. research. I enjoyed working together and I hope that we can keep this fruitful collaboration going for many years to come.

I thank all my colleagues of the mathematics department at the University of Twente for the many coffee and lunch discussions and the very good work environment that all of you create together. In particular, I would like to thank my roommates Jasper de Jong and Kamiel Cornelissen and, also, Bodo Manthey (whom I practically consider a roommate as well) for the daily conversations, both about work related and not so much work related topics.

For the design of the cover, my gratitude goes to my brother, Piet Hein. You did a great job of finding the cover I did not know I wanted.

I want to thank my mother for a childhood that has lead me to pursue a career in research. From going to several Ph.D. defenses at young age, to listening to discussions about your research on waste water treatment during dinner, I know I have a lot to thank you for, and this is certainly part of it.

Finally, Marije, thank you for enabling me to do what I do, and for being there for me no matter what.

Contents

CHAPTER 1

Introduction

1.1 Machine scheduling

Scheduling treats the choices that need to be made when given multiple tasks, such as “How do I perform the tasks?” and “In what order do I perform them?” While sometimes these decisions seem (or are) easy, at other times they are much more involved. Machine scheduling problems are a way of modeling these types of decisions.

In machine scheduling we are given a number of jobs (the tasks) and a number of machines. A *schedule* is an assignment of the jobs to the machines and an order in which the machines process those jobs. Both the jobs and the machines can have various properties. For example an order in which the job has to visit the machines, machine speeds, job processing requirements, a deadline before which a job has to be finished or the resources needed to process the job. These properties may constrain the set of schedules that are feasible. An objective function tells us what, given the constraints of the model, our goal is. This might be to minimize the time at which the last job finishes processing, to maximize the number of jobs that are processed before their deadline or to simply find a feasible schedule.

We identify the jobs by their index and the set of jobs is denoted by $N \equiv \{1, \ldots, n\}$. If there are multiple machines we also identify them by their index and the set of all machines is denoted by $M \equiv \{1, \ldots, m\}$. In most cases the index j refers to a job, while the index i refers to a machine. For each pair of a machine and a job, (i, j), we denote by p_{ij} the *processing time* of Job j on Machine i. This is the time that Job j has to be processed if it is scheduled on Machine i. Many different scheduling models exist, most of which can be succinctly referred to with the three-field notation introduced by Graham et al. [27]. The three-field notation is $\alpha \,|\, \beta \,|\, \gamma$, where α denotes the properties of the machine(s), β denotes the properties of the job and γ denotes the objection function. In this thesis we only consider so-called *single stage* machine scheduling problems, where jobs only need to be processed once on one of the machines. The simplest versions of machine scheduling problems are those where only a single machine is involved. We call these *single machine scheduling problems.* In three-field notation these are referred to by a 1 in the α-field $(1 \,|\, \,|\,)$. If more than one machine is involved we call the problem a *multiple machine scheduling problem.*

Definition 1.1 (Identical, related and unrelated machines)**.** We speak of *related* *machines* (Q | |) if each machine has a speed, s_i, and each job has a processing requirement, p_j, such that $p_{ij} = p_j/s_i$. *Identical machines* (P | |) refers to the model where for all jobs the processing time is independent from the machine on which it is processed, i.e. all machines have speed $s_i = 1$. In the case of *unrelated machines* (R | |) there is no restriction on the values of p_{ij} for any machine-job pair.

A Job j may also have a *weight*, w_j, which can indicate, for example, the importance of the job or the costs of letting that job wait one unit of time.

The focus of this thesis is on *utilitarian* objectives, by which we mean objectives that can be expressed as the sum of job utilities. The most common utilitarian objectives for scheduling are the sum of completion times and its weighted counterpart. Some other common objectives include: makespan, which is the finishing time of the whole schedule or, equivalently, the highest completion time among all jobs, and maximum or sum of lateness/tardiness/earliness. When jobs have due dates, the latter measure the deviation from those due dates.

1.1.1 Gantt charts

There are numerous ways to represent a schedule of jobs on a machine. For single machine scheduling we often use a vector of completion times, which gives for each job its completion time in the schedule. For a single machine and non-preemptive, deterministic schedules this uniquely determines the schedule. For multiple machines we additionally have to specify which machine processes which job.

A Gantt chart gives a clear visual representation of a schedule that can also be used for multiple machines. In a Gantt chart we visualize for each moment in time which job is being processed on which machine.

The following example shows for a simple scheduling problem a representation of a corresponding schedule as a Gantt chart.

Example 1.1. Consider the scheduling of three jobs on two machines. Each job can be processed on either machine. The processing time of Job 1 is 1 on either machine. The processing time of Job 2 is 1 on Machine 1 and 2 on Machine 2. The processing time of Job 3 is 3 on Machine 1 and 2 on Machine 2. Figure 1.1 is a Gantt chart of the schedule where Job 1 and Job 3 are scheduled, in that order, on Machine 1 and Job 2 is scheduled on Machine 2.

1.1.2 The scheduling polytope

While explicitly stating the order in which the jobs are processed might be a very clear and intuitive way to represent a schedule, there are also some disadvantages. This is especially the case when we are only interested in certain aspects of the schedule, such as when a certain job finishes processing or what the average waiting time of the jobs is. Since objectives often deal with this type of questions it can be convenient to look at a vector of completion times of the jobs to represent a schedule instead of an explicit ordering. Let $o : N \to N$ be an ordering of the jobs, such

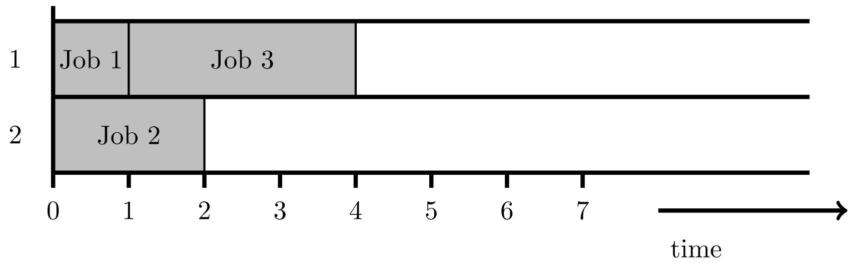


Figure 1.1: A Gantt chart of the schedule of Example 1.1.

that $o(j) = k$ means that Job j is the k-th job in the order. The completion times, $C \in \mathbb{R}^n$, can be computed as follows:

$$C_j = \sum_{\substack{k \in N, \\ o(k) \leq o(j)}} p_k \qquad \forall j \in N \ .$$

If instead we would be given a completion time vector C, the corresponding schedule is the one that processes the jobs in order of increasing completion times. These relations between the vector of completion times and orderings of the jobs show that indeed the completion time vectors are a useful way to represent schedules.

In this thesis, by the *start time* of a job we mean the moment it first starts its processing and by the *half time* of a job we mean the moment it has completed half of its processing requirement. We have the following relation between the start time S_j, the half time H_j and the completion time C_j of job j for non-preemptive schedules

$$C_j = H_j + \frac{1}{2}p_j = S_j + p_j \ .$$

In this thesis, for a given processing time vector p, we refer to the convex hull of all feasible start time vectors as the single machine scheduling polytope. The first full description of the single machine scheduling polytope was given in Queyranne [58]. For convenience of notation let

$$g(K) := \frac{1}{2}\left(\sum_{j \in K} p_j\right)^2 \ , \tag{1.1}$$

for any set $K \subseteq N$ of jobs.

Theorem 1.1 (Queyranne [58]). *The system of inequalities,*

$$\sum_{j \in K} S_j p_j \geq g(K) - \frac{1}{2}\sum_{j \in K} p_j^2 \qquad \text{for all } K \subset N \tag{1.2}$$

$$\sum_{j \in N} S_j p_j = g(N) - \frac{1}{2}\sum_{j \in N} p_j^2 \ , \tag{1.3}$$

fully describes the scheduling polytope for start times[1].

Note that the scheduling polytope can easily be shifted to describe the convex hull of all feasible completion time vectors or half time vectors.

The single machine scheduling polytope, is well understood [58]. In particular, it is known to be a polymatroid, and the separation problem, that decides if a given point is contained in the polytope or not, can be solved in $\mathrm{O}(n \log n)$ time. Also, its face lattice can be described by an ordered partition of N, as follows. Every $(n-k)$-dimensional face f of the scheduling polytope corresponds one-to-one with an ordered partition of N into k disjoint sets, $D_1, \dots, D_k$. With an ordered partition, we mean the (ordered) tuple $(D_1, \dots, D_k)$, with $D_i \cap D_j = \emptyset$ for all $i, j \in \{1, \dots, k\}$, $i \neq j$, and $\bigcup_{i=1}^{k} D_i = N$. The intended meaning is that inequalities (1.2) are tight for all $K_i := D_1 \cup \ldots \cup D_i$, $i \in \{1, \dots, k\}$. This corresponds to convex combinations of all schedules where jobs in K_i are scheduled before jobs in $N \setminus K_i$, for all $i \in \{1, \dots, k\}$. Furthermore, the schedules correspond to the ordered partitions $(\{\sigma(1)\}, \dots, \{\sigma(n)\})$ for all permutations σ. Each such ordered partition corresponds to a vertex of the scheduling polytope as follows: let $(\{\sigma(1)\}, \dots, \{\sigma(n)\})$ be an ordered partition and v the vertex it corresponds to, then

$$v_{\sigma(j)} = \sum_{i=1}^{j} p_{\sigma(i)} \qquad \text{for all } j \in N \ . \tag{1.4}$$

1.1.3 Linear orderings and permutations

Single machine scheduling is tightly connected to ordering. In absence of any idle time, any ordering of the jobs uniquely determines a schedule.

Definition 1.2 (Linear ordering polytope)**.** A *linear ordering* of n elements describes for any pair of elements, (i, j), either that i is ordered before j or j is ordered before i. Furthermore, if i is ordered before j and j is ordered before k, then also i is ordered before k. We represent such an ordering as a vector $\delta \in [0,1]^{n^2}$, that satisfies the following system of inequalities:

$$\begin{aligned} \delta_{kj} + \delta_{jk} &= 1 && \forall j,k \in N, j \neq k && (1.5)\\ \delta_{\ell k} + 1 &\geq \delta_{\ell j} + \delta_{jk} && \forall j,k,\ell \in N && (1.6)\\ \delta_{jk} &\in \{0,1\} && \forall j,k \in N \ , && (1.7) \end{aligned}$$

where $\delta_{kj} = 1$ denotes that k is ordered before j and $\delta_{kj} = 0$ otherwise.

The *linear ordering polytope* is the convex hull of all such vectors.

To any linear ordering we can relate the schedule that schedules all jobs in that order. Any vertex of the scheduling polytope then corresponds to exactly one such

[1]If $p_j > 0$ for all $j \in N$, none of these inequalities is redundant, and the dimension is $n-1$ [58]. Note that, for the degenerate case, where $p_k = 0$ for some jobs k, we would have to add constraints $0 \leq C_k \leq \sum_{j \in N} p_j$ in order to describe the convex hull of schedules.

ordering. For such a vertex we get the start time S_j of any job, j, as follows:

$$S_j = \sum_{k \in N \setminus \{j\}} \delta_{kj} p_k \qquad \forall j \in N \ . \tag{1.8}$$

Definition 1.3 (Permutahedron)**.** The *permutahedron* is the polytope that consists of all convex combinations of vectors of permutations of $\{1, \dots, n\}$. The permutahedron is a special case of the scheduling polytope for completion times where $p_j = 1$ for all $j \in N$.

We can interpret a permutation vector as an ordering of the jobs. If we do this, it is easy to compute from a vertex of the permutahedron the corresponding start time vector.

1.1.4 Smith's rule

For the single machine scheduling problem with sum of weighted completion times objective $(1 \,|\,|\, \sum w_j C_j)$, Smith [66] describes what is known as Smith's rule or the *weighted shortest processing time first* (WSPT) rule. It is processing the jobs in descending order of the ratio of weight over processing time.

Theorem 1.2 (Smith's rule [66])**.** *On a single machine, a schedule is optimal for the sum of weighted completion time objective if and only if it schedules the jobs in WSPT order.*

In particular, for the non-weighted case ($w_j = 1$ for all j) this means that a schedule is optimal if and only if it schedules the jobs in *shortest processing time first* (*SPT*) order.

1.2 Algorithmic game theory

Algorithmic game theory treats decision making in settings where one or more players, who each make their own strategic decisions, are involved. In this thesis the games we treat are so-called scheduling games, where the players correspond to jobs in a machine scheduling setting. We identify these n players by there index and the set of all players is denoted by $N \equiv \{1, \dots, n\}$. We consider non-cooperative games, where we assume that every Player j has its own utility function, $u_j(\cdot)$, which it tries to maximize, and a set of possible actions, Σ_j, which we call strategies. The utilities of the players are functions of the strategy vector, which consists of one strategy for each player.

Definition 1.4 (Non-cooperative game)**.** A game is a triple $G = (n, u, \Sigma)$, where $\Sigma = \Sigma_1 \times \ldots \times \Sigma_n$ and u is the vector of utility functions of the players.

If $\sigma \in \Sigma$ is a strategy vector then σ_j denotes the strategy of Player j, σ_{-j} denotes the strategies of all players except Player j and (σ_j, σ_{-j}) denotes the whole strategy vector. If a player chooses a single strategy we refer to it as a *pure strategy.* In general we allow players to choose a probability distribution over such pure strategies. We call such a probability distribution a *mixed strategy.* A vector σ of possibly mixed strategies is called a mixed strategy vector.

Each combination of strategies chosen by the players leads to an *outcome* of the game. This outcome is represented by the utilities of the players. Given a game we are looking to predict what strategies the players will choose and compare the corresponding outcome to a desirable optimal outcome. For scheduling games the strategies of the players determine a schedule and the players utilities can be computed as a function of their completion time in the schedule. We illustrate some of the concepts of algorithmic game theory with scheduling games as examples.

1.2.1 Equilibria

Since players make their own decisions in non-cooperative games, not every strategy vector is viable. There are several notions that describe what a viable strategy vector may look like. We call these outcomes *equilibria.*

The concept of *Nash equilibria* (*NE*) was introduced by Nash [52] as a solution concept for non-cooperative games. It is based on the assumption that players will always try to improve their utility. Therefore, a strategy vector is considered to be 'stable' if no player can unilaterally improve.

Definition 1.5 (Nash equilibrium)**.** A (mixed) strategy vector, ν, is a Nash equilibrium if for any Player $j \in N$ and any strategy ν'_j of Player j

$$\mathbb{E}_{\sigma\sim(\nu_j,\nu_{-j})}[u_j(\sigma)] \geq \mathbb{E}_{\sigma\sim(\nu'_j,\nu_{-j})}[u_j(\sigma)] \ , \tag{1.9}$$

where $\mathbb{E}_{\sigma\sim\nu}$ denotes the expectation over the stochastic variable strategy vector σ, such that σ_j is distributed according to ν_j for all j.

Any strategy, σ_j, that maximizes a Player j's expected utility, for a given strategy vector, ν_{-j}, is called a best response of j with respect to ν_{-j}. In the simple case where ν is a pure strategy vector we have, instead, for Player $j \in N$ and any $\nu'_j \in \Sigma_j$,

$$u_j(\nu_j, \nu_{-j}) \geq u_j(\nu'_j, \nu_{-j}) \ .$$

We refer to this case as *pure Nash equilibrium* (*PNE*) and similarly to the mixed strategy case as *mixed Nash equilibrium* (*MNE*).

Example 1.2 (Nash equilibrium)**.** Consider scheduling three jobs on two identical machines. Jobs 1 and 2 both have a processing time of 1 and Job 3 has a processing time of 2. Each machine schedules the jobs assigned to it in SPT order and breaks ties in favor of Job 1. The jobs try to minimize their completion time. For this instance, both assignments in Figure 1.2 are pure Nash equilibria, since for no job changing the machine it is processed on, improves its completion time.

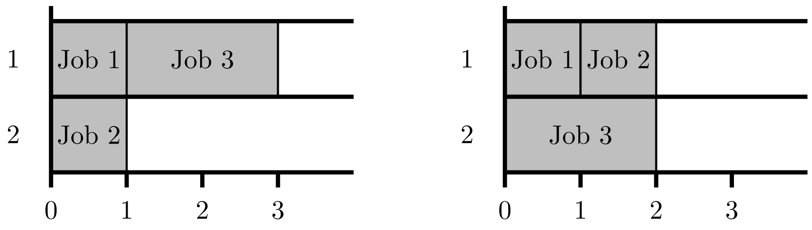


Figure 1.2: Two Nash equilibria for the same scheduling game.

One disadvantage of pure Nash equilibria is that not every game induces such an equilibrium. Example 1.3 shows a matching pennies game where no pure Nash equilibrium exists.

Example 1.3. Two people, Player A and Player B, play a game of matching pennies. The strategies of both players consist of choosing heads or tails. If both choices match, Player A wins, otherwise Player B wins. If a player wins he receives a utility of 1, otherwise his utility is -1. Now, it is easy to see that if Player A plays tails, Player B has an incentive to play heads. Likewise, if Player A plays heads, Player B has an incentive to play tails. Similar reasoning holds when Player B plays heads.We see that no combination of pure strategies leads to a Nash equilibrium.

Nash [52] shows that allowing the players to play mixed strategies solves this disadvantage.

Theorem 1.3 (Nash [52]). *For every finite non-cooperative game there is a mixed strategy vector ν that satisfies* (1.9).

The game in Example 1.3 has one mixed Nash equilibrium, where both players play head and tails both with probability 1/2. In that case both players have probability 1/2 to win and can not improve by changing their strategy.

Correlated equilibria and *coarse correlated equilibria* generalize upon (mixed) Nash equilibria. In this thesis we do not explicitly treat these equilibria. However, for sake of completeness and since the smoothness framework as described below does imply bounds that involve these equilibria as well, we give definitions below.

Definition 1.6 (Correlated equilibrium). A probability distribution ν on Σ is a correlated equilibrium if for each Player j and for all $\sigma_j^* \in \Sigma_j$, such that $\mathbb{P}_{\sigma\sim\nu}[\sigma_j = \sigma_j^*] > 0$ and all $\sigma_j' \in \Sigma_j$ we have

$$\mathbb{E}_{\sigma\sim\nu}[u_j(\sigma_j, \sigma_{-j})|\sigma_j = \sigma_j^*] \geq \mathbb{E}_{\sigma\sim\nu}[u_j(\sigma_j', \sigma_{-j})|\sigma_j = \sigma_j^*] \ .$$

Here $\mathbb{E}_{\sigma\sim\nu}$ denotes the expectation over the stochastic variable strategy vector σ that is distributed according to ν.

Note that, in general, a distribution on Σ can not be represented by a mixed strategy vector.

Definition 1.7 (Coarse correlated equilibrium)**.** A probability distribution ν on Σ is a coarse correlated equilibrium if for each Player j and for all $\sigma'_j \in \Sigma_j$ we have

$$\mathbb{E}_{\sigma\sim\nu}[u_j(\sigma_j,\sigma_{-j})] \geq \mathbb{E}_{\sigma_{-j}\sim\nu_{-j}}[u_j(\sigma'_j,\sigma_{-j})] \ .$$

Here $\mathbb{E}_{\sigma\sim\nu}$ denotes the expectation over the stochastic variable strategy vector σ that is distributed according to ν.

For a game G let PNE(G) denote the set of pure Nash equilibria, NE(G) the set of (mixed) Nash equilibria, CE(G) the set of correlated equilibria and let CCE(G) denote the set of coarse correlated equilibria. Then the following inclusions hold

$$\mathrm{PNE}(G) \subseteq \mathrm{NE}(G) \subseteq \mathrm{CE}(G) \subseteq \mathrm{CCE}(G) \ .$$

1.2.2 Price of anarchy

The *price of anarchy* (*POA*) [45, 55] is a measure that compares the worst case Nash equilibrium of a game to the optimal solution in the corresponding optimization problem. One could say that the price of anarchy measures the deterioration of system performance due to selfishness of the players and the lack of central coordination. Here, the metric for the quality of a solution is in terms of the central objective function. In the economic literature the central objective function is rather called social choice function [51]. In this thesis we consider cost minimization games, where the central objective function is to minimize some cost function. We therefore define the price of anarchy for those games, as follows. Equivalent definitions exist for maximization games [45, 55].

Definition 1.8 (Price of anarchy for a game)**.** The price of anarchy for a game, G, is defined as

$$\mathrm{POA(G)} = \frac{\max_{\nu\in\mathrm{NE(G)}}\mathrm{Cost}(\nu)}{\mathrm{Cost}(\mathrm{OPT}(G))} \ ,$$

where OPT(G) is an optimal solution of game G and Cost(σ) denotes the central objective function value in outcome σ.

We also use the price of anarchy in a more general sense to compare Nash equilibria to optimal solutions for a set of games.

Definition 1.9 (Price of anarchy for a set of games)**.** For a class of games Γ the price of anarchy is defined as

$$\mathrm{POA}(\Gamma) = \sup_{G\in\Gamma}\mathrm{POA(G)} \ .$$

The price of anarchy can similarly be defined for other equilibria. Most notably the *pure price of anarchy* (*PPOA*) is used to refer to the case where only pure Nash equilibria are considered. The framework discussed in the next section provides bounds on the price of anarchy for all of the in Section 1.2.1 discussed equilibria.

1.2.3 Smoothness

In Roughgarden [63] the *robust price of anarchy* is introduced as an upper bound on the price of anarchy for pure and mixed Nash equilibria, correlated equilibria and coarse correlated equilibria.

Definition 1.10 ((λ, μ)-smooth games [63])**.** A utilitarian cost-minimization game is (λ, μ)-*smooth* if for every two strategy vectors ν and σ,

$$\sum_{j=1}^{n} \mathrm{Cost}_j(\sigma_j, \nu_{-j}) \leq \lambda \cdot \mathrm{Cost}(\sigma) + \mu \cdot \mathrm{Cost}(\nu) \ , \tag{1.10}$$

where $\mathrm{Cost}(\sigma)$ denotes the utilitarian objective value for strategy vector σ, i.e. the sum of the utilities of the agents.

If a utilitarian game is (λ, μ)-smooth with $\lambda > 0$ and $\mu < 1$, it follows that for any (mixed) Nash equilibrium ν and optimal solution σ

$$\sum_{j=1}^{n} \mathrm{Cost}_j(\nu) \leq \sum_{j=1}^{n} C_j(\sigma_j, \nu_{-j}) \leq \lambda \cdot \mathrm{Cost}(\sigma) + \mu \cdot \mathrm{Cost}(\nu) \ . \tag{1.11}$$

From (1.11) it follows directly that $\frac{\lambda}{1-\mu}$ is an upper bound on the price of anarchy for any (λ, μ)-smooth game. This bound also holds for correlated equilibria and coarse correlated equilibria [63]. The *robust price of anarchy* is defined in Roughgarden [63] as the least upper bound on the price of anarchy that is provable through a smoothness argument.

Definition 1.11 (Robust price of anarchy [63])**.** The robust price of anarchy of a cost-minimization game is

$$\inf \left\{ \frac{\lambda}{1-\mu} \middle| \text{the game is } (\lambda, \mu)\text{-smooth} \right\} \ .$$

In Chapter 2 we also use the notions of *semi-smoothness* and *niceness* of games.

Definition 1.12 ((λ, μ)-semi-smooth games [2])**.** A cost-minimization game is (λ, μ)-*semi-smooth* if for every Player j there is a mixed strategy σ_j such that for each strategy vector ν,

$$\sum_{j=1}^{n} \mathbb{E}_{\eta_j \sim \sigma_j}[\mathrm{Cost}_j(\eta_j, \nu_{-j})] \leq \lambda \cdot \mathrm{Cost}(\sigma^*) + \mu \cdot \mathrm{Cost}(\nu) \ , \tag{1.12}$$

where σ^* is an optimal solution.

Semi-smoothness is a generalization of regular smoothness that still provides bounds for (mixed) Nash equilibria, correlated equilibria and coarse correlated equilibria in the same fashion as regular smoothness [2].

Definition 1.13 ((λ, μ)-nice games [2])**.** A cost-minimization game is (λ, μ)-*nice* if for every mixed strategy vector ν there is a mixed strategy vector σ such that

$$\sum_{j=1}^{n} \mathrm{Cost}_j(\sigma_j, \nu_{-j}) \leq \lambda \cdot \mathrm{Cost}(\sigma^*) + \mu \cdot \mathrm{Cost}(\nu) \ , \tag{1.13}$$

where σ^* is an optimal solution.

Niceness does not imply the same bounds as semi-smoothness. However it is easy to see that it still provides upper bounds on the price of anarchy.

1.2.4 Mechanism design

All of the previous examples and definitions for optimization problems and algorithmic game theoretical problems assume that we are given full information on which we need to base our decision. More specifically, we assume that this information is perfect and complete, even if we are dealing with selfish players. In mechanism design problems we partly drop this assumption, by assuming that players have private information that is relevant for the outcome of the game. Mechanism design can be seen as designing a game such that the result is beneficial for the designer.

We start with a combinatorial optimization problem which has a certain set of outcomes, A, which we call *allocations*. We introduce players with some private information, about which they are allowed to lie. In this setting every player $j \in N$ has private information called its *type*, t_j. This type consists of one or multiple parameters and we assume that for each player there is a type set, T_j, and a probability distribution over that type set, $\varphi_j : T_j \to [0, 1]$, that are publicly known. Moreover, each player has a valuation function $v_j : A \times T_j \to \mathbb{R}$. The *mechanism designer* decides on a set of strategies for each player, that determine the allocation to all the players, and a vector of payments, π. The utility for Player j is then $u_j = v_j + \pi_j$. The mechanism designer tries to maximize his own utility function, which in our case is minimizing the sum of payments made to the players.

In the resulting game the players do not simply choose one strategy, but they choose one strategy for every one of their possible types. If, in expectation with respect to the type distribution and the chosen strategies of the other players, no player can improve his utility by unilaterally changing his strategy, we call the played strategies a *Bayes-Nash equilibrium*. If this is true for any realization of the strategies and types of the other players, we call the player strategies a *dominant strategy equilibrium*.

Example 1.4 (Scheduling mechanism design)**.** Consider the scheduling of two jobs on one machine. Each of the jobs has a processing time, p_1, p_2, and a weight, w_1, w_2. The cost for waiting for a job j is $w_j S_j$, where S_j is the jobs start time. The owner of the machine needs to compensate the jobs for their waiting costs.

The processing times of the jobs are known and both the same: $p_1 = p_2 = 1$. However, the weights are uncertain: Job 1 has weight $w_1 = 4$ and Job 2 either has weight $w_2 = 1$ or it has weight $w_2 = 3$, each with probability 0.5. If the types are not private it is easy to see that it is optimal to always schedule Job 1 first. This results in an expected sum of weighted start times equal to 2.

Now consider that the type of each job is private information. Suppose that we simply ask the jobs what their type is. We still know that Job 1 has weight $w_1 = 4$. However, Job 2 may represent itself as having $w_2 = 3$, while its type is actually $w_2 = 1$. If it does, Job 2 is scheduled last and receives a payment of 3, while its real waiting costs are only 1. So in this case it would be beneficial for Job 2 to lie about its true type. Furthermore the expected payments made to the jobs in this case would be 3. By changing the schedule when Job 2 reports type $w_2 = 3$, such that Job 2 is scheduled first in that case, Job 2 does not benefit from lying when it has type $w_2 = 1$. This results in expected payments made to the jobs equal to 2.5.

Direct revelation mechanisms

In this thesis we restrict ourselves to *direct revelation mechanisms* in which a player's strategy space for each of their types is to report one type. Myerson's revelation principle tells us that in many cases this is a valid simplification [49]. In particular for the model we study in Chapter 3 the revelation principle holds.

Theorem 1.4 (Revelation principle [49]). *For any mechanism that has a Bayes-Nash equilibrium there exists an equivalent feasible direct revelation mechanism for which a Bayes-Nash equilibrium exists and which gives to the mechanism designer and all players the same expected utilities as in the given mechanism.*

In Theorem 1.4, one can replace the Bayes-Nash equilibrium by another equilibrium concept, while remaining true. In particular, this is the case for the dominant strategy equilibrium.

Since, for direct revelation mechanisms, the strategy space of all players is simply to report a type, we define a direct revelation mechanism as a tuple (f, π), where f is an *allocation rule* and π is a payment vector. For each type vector $t \in T = T_1 \times \ldots T_n$, the allocation rule is a function, $f : T \to A$, that decides which allocation is chosen. The payment vector is a function that for each type vector $t \in T$ and each player $j \in N$ specifies a payment, $\pi_j : T \to \mathbb{R}$, made to Player j.

We consider scheduling mechanism design in which an allocation rule is assigning schedules to type vectors and the valuation of the players is based on their (expected) start time in those schedules. We refer to the allocation rule for such a scheduling game as scheduling rule. For type vectors we use the same notation as for strategy vectors, namely if $t \in T$ is a type vector then t_j denotes the type of Player j, t_{-j} denotes the types of all players except Player j and (t_j, t_{-j}) denotes the whole type vector. Likewise, we denote with $\varphi(t)$ and $\varphi_{-j}(t_{-j})$ the probability of respectively t and t_{-j} occurring and with T_{-j} the set of all possible vectors that exclude Player j.

The scheduling rule f directly implies (expected) start times for the players. We therefore write the (expected) start time of Player j for type vector t as $S_j(f, t)$ and the expected start time of Player j when reporting type t_j as $ES_j(f, t_j)$.

$$ES_j(f, t_j) = \sum_{t_{-j} \in T_{-j}} \varphi_{-j}(t_{-j}) S_j(f, (t_j, t_{-j})) \ .$$

For the models that we study in this thesis the utility of Player j is

$$u_j(t_j, f, t) = -w_j(t_j) S_j(f, t) + \pi_j(t) \ ,$$

where $w_j(t_j)$ is the weight that Job j has in type t_j. Note that the corresponding valuation function for Player j is $v_j(t_j) = -w_j(t_j) S_j(f, t)$.

Incentive compatibility and individual rationality

A mechanism, (f, π), is *Bayes-Nash incentive compatible* (*BNIC*) if no player has an incentive to lie about their type in expectation.

Definition 1.14 (Bayes-Nash incentive compatibility)**.** A mechanism, (f, π), is Bayes-Nash incentive compatible if for each Player j and any pair of types $t_j, t'_j \in T_j$

$$\sum_{t_{-j} \in T_j} \varphi_{-j}(t_{-j}) u_j(t_j, f, (t_j, t_{-j})) \geq \sum_{t_{-j} \in T_j} \varphi_{-j}(t_{-j}) u_j(t_j, f, (t'_j, t_{-j})) \ .$$

For the scheduling mechanism design problems we consider, a mechanism is BNIC if for each Player j and any pair of types $t_j, t'_j \in T_j$

$$- w_j(t_j) ES_j(f, t_j) + E\pi_j(t_j) \geq -w_j(t_j) ES_j(f, t'_j) + E\pi_j(t'_j) \ , \tag{1.14}$$

where $E\pi_j(t_j) = \sum_{t_{-j} \in T_j} \varphi_{-j}(t_{-j}) \pi_j(t_j, t_{-j})$. We immediately see that, when considering BNIC mechanisms it suffices to have payments made to each job solely dependent on the type of that job (and not on the whole type vector). We say that an allocation rule f is *Bayes-Nash implementable* if there exist payments π such that the mechanism is BNIC.

Similarly, we can define *dominant strategy incentive compatible* (*DSIC*) mechanisms. These are mechanisms where no matter what the strategy of other players is, no player has an incentive to lie about their type.

Definition 1.15 (Dominant strategy incentive compatibility)**.** A mechanism is dominant strategy incentive compatible if for each Player j and any pair of types $t_j, t'_j \in T_j$ and all vectors $t_{-j} \in T_{-j}$

$$u_j(t_j, f, (t_j, t_{-j})) \geq u_j(t_j, f, (t'_j, t_{-j})) \ .$$

For the scheduling mechanism design problems we consider, a mechanism is DSIC if for each Player j and any pair of types $t_j, t'_j \in T_j$ and all vectors $t_{-j} \in T_{-j}$

$$-w_j(t_j) S_j(f, (t_j, t_{-j})) + \pi_j(t_j, t_{-j}) \geq -w_j(t_j) S_j(f, (t'_j, t_{-j})) + \pi_j(t'_j, t_{-j}) \ .$$

An allocation rule f is *dominant strategy implementable* if there exist payments π such that the mechanism is dominant strategy incentive compatible.

In addition to incentive compatible, we want that a mechanism is *individually rational* (*IR*). This property implies that no truthful report of a type results in a negative utility for any of the players. In the Bayes-Nash scheduling mechanism design setting this translates to

$$\pi_j^i - w_j^i ES_j^i \geq 0 \ ,$$

for all jobs j and all types $t_j \in T_j$.

1.3 Algorithm analysis and complexity

In this thesis we design and discuss several algorithms. To this end it is useful to introduce some concepts of algorithm analysis and complexity theory. This section is in no way meant to be comprehensive on this subject. We refer the interested reader to Garey and Johnson [25] and Papadimitriou [54]. This section is mostly based on Chapter 8 and Chapter 15 of Papadimitriou and Steiglitz [56].

1.3.1 Time complexity and input size

An algorithm is a prescribed sequence of instructions such that a computer would be able to execute them. This is a very loose way to describe what an algorithm is, but it captures the essence. Turing [67] describes the *Turing machine* as a tool to mathematically analyze the termination and *running time* of algorithms. The running time is a measure for how long it takes an algorithm to terminate. Of course, this is dependent on the exact computer on which we run the algorithm, and therefore the theoretical running time is often expressed in terms of the number of elementary operations. That is, arithmetic operations, comparisons, branching, and so on. We assume that these elementary operations take unit time and that the speed of a computer is linear in these time units, i.e. we can say one computer is ten times as fast as another, meaning that it performs ten times a many elementary operations in the same time frame.

For most algorithms the running time is dependent on both the input and its representation. For example, it takes more time to sort a sequence of 10 integers than it takes time to sort a sequence of only two integers, and an algorithm with a graph as its input, may take more time when this graph is represented as adjacency lists as opposed to a representation as an adjacency matrix.

Example 1.5 (Bubble sort)**.** We want to order n jobs in SPT order. Let us assume that the jobs are given in a list of n integer numbers which represent their processing times. We apply the following algorithm.

Compare the first pair of adjacent integers. If they are in the right order move to the next pair (the highest integer of the current pair and the next integer), otherwise swap them and move to the next pair. Do this until the end of the list is reached and start over at the first pair. Repeat this process until no swaps are made on any pair of the list.

Suppose we want to sort two integers with the above algorithm and suppose that they are in reverse order. Then the algorithm compares the two integers once and swaps them, then it compares them another time and terminates. So it takes two comparisons and one swap to sort the list of two elements that are in reverse order. Now if we start with a list with ten elements in reverse order, it clearly takes more time to sort them using this algorithm. It takes even more time to sort a hundred elements.

In the example of sorting a list of n integers with the bubble sort algorithm it takes n comparisons and at most n swaps to go through the list once and every time the algorithm passes through the list, one more element ends up in the correct place in the list. Therefore the algorithm has to go through the list at most n times. So it takes no more than n^2 comparisons and no more than n^2 swaps and in total no more than $2n^2$ elementary operations. Instead of mentioning for each algorithm the exact number of number of elementary operations that is needed, we use the *rate of growth* of the running time of an algorithm. *Big O notation* is a useful instrument to express the worst case running time of an algorithm.

Definition 1.16 (Big O notation)**.** Let $f(n)$ and $g(n)$ be two functions from the positive integer numbers to the positive real numbers. Then $f(n) = \mathrm{O}(g(n))$ if for some $c \in \mathbb{R}$ and for all n, large enough, $f(n) \leq cg(n)$.

Using big O notation we can now say that, worst case, bubble sort takes time $\mathrm{O}(n^2)$ to sort a list of n integers. We say that the *time complexity* of bubble sort is $\mathrm{O}(n^2)$.

To analyze the time complexity of an algorithm we compare it to the *size of the input*. That is, the number of symbols needed to encode the input in a computer. For common arithmetic systems, decimal or binary for example, an integer k can be encoded in $\mathrm{O}(\log k)$ symbols. Note that the used base of the logarithm is irrelevant since $\log_b(k) = \log(k)/\log(b)$ and $\log(b)$ is a constant for any fixed base b. Since computers regularly use a fixed number of bits to represent any integer, independent of its size, we treat the encoding length for an integer as a constant[2]. We can therefore say that a list of n integers can be encoded in size $\mathrm{O}(n)$.

Definition 1.17 (Polynomial time algorithm)**.** Let A be an algorithm with input size $\mathrm{O}(n)$. We say A is a *polynomial time algorithm* if its time complexity is $\mathrm{O}(n^k)$ for some k independent of the input.

In terms of the maximal size instances a computer can solve using a certain algorithm polynomial time algorithms are in general considered efficient. Of course there is a difference in efficiency between an algorithm that takes $\mathrm{O}(n)$ (linear) time

[2]Note that there are cases where the representation of a number on a computer is influential on size of the input. In particular, this is the case when really large numbers are involved (large enough that their encoding size might exceed the encoding size of the rest of the input).

and one that takes $O(n^{10})$ time. However, the difference between a polynomial time algorithm and algorithms of which the time complexity can not be bounded by a polynomial is much more significant. Such algorithms may for example take exponential running time, e.g. $O(2^n)$. Example 1.6 illustrates the difference between polynomial time algorithms and exponential time algorithms for hardware speedups.

Example 1.6. Suppose two algorithms, A and B, have the same input, which has size n. Algorithm A is a polynomial time algorithm with time complexity $O(n^2)$. Algorithm B is an exponential time algorithm with time complexity $O(2^n)$. Now suppose we have an instance large enough that both algorithms show their asymptotic behavior in running time. Let us say that the instance takes both algorithms one hour to complete on our current computer. However, we want to solve larger inputs within one hour. So, we buy a new computer that is 100 times as fast as the old one. Now from Algorithm A we can expect that it solves inputs about ten times as large in one hour on this new computer. Algorithm B however only allows an additive increase of the input size of about $\log_2 100 \approx 6.64$.

1.3.2 Complexity classes $\mathcal{P}$ and $\mathcal{NP}$

With the tools from the previous section we can describe the complexity of an algorithm. However, this does not tell us anything about how well the algorithm does for the purpose it was designed for. For example, while bubble sort has a time complexity of $O(n^2)$, there also exist sorting algorithms that have time complexity $O(n \log n)$. Meanwhile there are also many problems for which no polynomial time algorithms are known. Therefore we need to be able to asses the complexity of a problem, how hard a problem is to solve. To discuss this in a meaningful way we use *polynomial time reductions* and complexity classes. The idea is that each complexity class specifies a bound on the complexity of the problems in that class.

First let us define what we mean by a problem.

Definition 1.18 (Minimization, maximization and decision problem)**.** A *minimization problem* is a triple $(\mathcal{I}, F, c)$, where $\mathcal{I}$ is a set of instances; given an instance $I \in \mathcal{I}$, $F(I)$ is the set of feasible solutions; and given an instance $I \in \mathcal{I}$ and a feasible solution $x \in F(I)$, $c(I,x)$ is a cost function.

For a given instance $I \in \mathcal{I}$ the problem is to find a feasible solution $x \in F(I)$ such that

$$c(I,x) \leq c(I,x') \qquad \text{, for all } x' \in F(I) \ .$$

A *maximization problem* is a triple $(\mathcal{I}, F, c)$, defined the same as a minimization problem except that given an instance $I \in \mathcal{I}$ the problem is to find a feasible solution $x \in F(I)$ such that

$$c(I,x) \geq c(I,x') \qquad \text{, for all } x' \in F(I) \ .$$

Given a minimization problem $(\mathcal{I}, F, c)$ the corresponding *decision problem* is a triple $(\mathcal{I}', F, c)$, where $\mathcal{I}' = \mathcal{I} \times \mathbb{R}$ and given $(I, C) \in \mathcal{I}'$, the question is: "Does there exist a feasible solution $x \in F(I)$ such that $c(I,x) \leq C$?"

Decision versions for maximization problems are defined similarly.

Decision problems are defined by the fact that the answer is either “yes” or “no.” Some problems are already decision problem to begin with, such as the Hamiltonian cycle problem or the Satisfiability problem. We call an instance of a decision problem to which the answer is “yes” a *yes-instance.* Likewise, we call an instance of a decision problem to which the answer is “no” a *no-instance.*

We say a problem is *polynomial time solvable* if there is a polynomial time algorithm that solves that problem correctly. Here, solving correctly means, for a decision problem, returning “yes” for a yes-instance and “no” for a no-instance, while for an optimization problem it means returning a feasible solution that optimizes the objective function. The complexity class $\mathcal{P}$ consists of all problems that are polynomial time solvable.

Definition 1.19 (Polynomial time reduction (Karp reduction [44]))**.** There is a polynomial time reduction from decision problem A to decision problem B if and only if there is a polynomial time algorithm that translates any yes-instance of problem A into a yes instance of problem B and any no-instance of problem A into a no-instance of problem B.

Theorem 1.5. *If for two decision problems A and B, there is a polynomial time reduction from A to B and B is polynomial time solvable, then A is also polynomial time solvable.*

Proof. Suppose I_A is an instance for Problem A we use the exiting polynomial time algorithm to translate this instance to I_B an instance for Problem B. We now use the existing polynomial time algorithm to solve I_B, which answers us either “no” or “yes.” Since the reduction translates yes-instances to yes-instances and no-instances to no-instances, the answer to I_B for Problem B is the answer to I_A for Problem A. □

Definition 1.20 (Complexity class $\mathcal{NP}$)**.** A decision problem P is in the complexity class $\mathcal{NP}$ if there is a polynomial time algorithm A such that an instance I of P is a yes-instance if and only if there is a polynomial size certificate $q(I)$ such that Algorithm A verifies (outputs “yes”) $q(I)$ in polynomial time.

A generally made assumption is that $\mathcal{P} \neq \mathcal{NP}$ and therefore that there are computationally hard problems in $\mathcal{NP}$. This justifies seeking non-exact (approximation) algorithm for problems that are *$\mathcal{NP}$-hard.*

Definition 1.21 ($\mathcal{NP}$-hard problem)**.** A Problem A is $\mathcal{NP}$-hard if for any Problem B in $\mathcal{NP}$ there is a polynomial reduction to A.

If a problem in $\mathcal{NP}$ is $\mathcal{NP}$-hard, we call it *$\mathcal{NP}$-complete.*

A method that is often used in practice to solve optimization problems is translating the problem to a *linear programming* (*LP*) problem.

Definition 1.22 (Linear programming (LP))**.** Linear programming is the minimization problem $A \in \mathbb{R}^{m\times n}$ $b \in \mathbb{R}^m$ $c \in \mathbb{R}^n$ that solves

$$\max_{x\in\mathbb{R}^n}\{cx \mid Ax \leq b\} \ .$$

Linear programming is a theoretically attractive way to formulate problems since it is polynomial time solvable by the ellipsoid method [28].

Theorem 1.6 ([28])**.** *Linear programming is polynomial time solvable.*

We also distinguish *integer linear programming* (*ILP*), where $x \in \mathbb{Z}^n$, and *mixed integer linear programming* (*MIP*), where for some set $K \subset \{1,\dots,n\}$ we have $x_k \in \mathbb{Z}$ for $k \in K$ and $x_k \in \mathbb{R}$ for $k \notin K$. Under the assumption that $\mathcal{P} \neq \mathcal{NP}$, these problems are not polynomial time solvable [56].

1.4 Thesis outline

In Chapter 2, we analyze the price of anarchy for the classical related machine scheduling problem, to minimize the total completion time. While the main focus is on the SPT rule (shortest processing time first) as scheduling rule, we also discuss some other possible scheduling rules. The main result is an upper bound of 2 for the price of anarchy, if SPT is used as the scheduling rule. We also give a lower bound of $e/(e-1) \approx 1.58$. The upper bound proof is based on a new characterization of the optimal solution, which is interesting in its own right. Both the lower and the upper bound hold for pure Nash equilibria, mixed Nash equilibria, correlated equilibria and coarse correlated equilibria. Most of the results from Chapter 2 are published as [34].

In Chapter 3, we consider a private information setting. We address the design of optimal mechanisms for a single machine scheduling problem. In this setting, both the weight and the processing time, are private to the jobs. Assuming that jobs need to be compensated for waiting, and the cost of waiting is governed by the private weight of a job, the goal is to find a mechanism that minimizes the total expected payment that is made to the jobs.

The problem where only weights are private can be solved in polynomial time. We settle the complexity of the problem with private weights and processing times, or in other words, with multi-dimensional types. We show that a randomized optimal mechanism can be computed in polynomial time. Our result is obtained by linear programming techniques, and at its core we show that an exponential size LP relaxation of the problem can be projected to a polynomial size LP, without any loss in performance. The result is an LP that computes a so-called interim solution, in our case expected start times of jobs. The final step is then to translate this interim solution to a randomized mechanism. This requires a decomposition of a point in the scheduling polytope into a convex combination of vertices, which can be done efficiently too. The results from Chapter 3 are published as [35, 36].

With the above, we settle the complexity of the problem to compute an optimal randomized mechanism. However, it is not clear if and how the procedure could be derandomized, and indeed, the complexity of computing an optimal deterministic mechanism remains open. It is not even clear if the corresponding decision problem belongs to the class $\mathcal{NP}$. Therefore, we consider in Chapter 4 the same problem as in Chapter 3, where we additionally impose a condition called IIA, independence of irrelevant alternatives. For the single machine scheduling problem, it requires that the relative order of any two jobs must only depend on the types of these jobs, but not on the types of any other job. When solutions satisfy the IIA condition, the problem falls in the class $\mathcal{NP}$, because any such mechanism can be represented as a linear ordering of the types across all jobs. We show how this can be exploited algorithmically, and present results using local search and other constructive methods to compute deterministic IIA mechanisms. In computational experiments, we confirm that these methods are fast, and have the potential to compute close to optimal mechanisms, even for very large scale instances that are hard to tackle with the LP-based techniques of Chapter 3. The results from Chapter 4 are found in [39].

In Chapter 5, we address the algorithmic problem to decompose a given point in the scheduling polytope into a convex combination of vertices. While this problem arises in the mechanism design context of Chapter 3, it turns out to be of interest in its own right. A polynomial time algorithm follows from standard techniques whenever the separation problem for a given polytope can be solved in polynomial time. However that does not necessarily yield combinatorial algorithms. A related problem is to efficiently compute the intersection of a polytope with a line. For that problem, we give a, combinatorial, time $\mathrm{O}(n^2 \log n)$ algorithm for the scheduling polytope. From that, a time $\mathrm{O}(n^3 \log n)$ algorithm follows for the decomposition problem. The main result of this chapter, however, is a time $\mathrm{O}(n^2)$ algorithm to solve the decomposition problem for the scheduling polytope. The algorithm exploits the geometry of the scheduling polytope, and crucially uses the fact that the scheduling polytope is a zonotope, i.e. that all its faces are centrally symmetric. From that, we derive a simple description of the barycentric subdivision of the scheduling polytope, and show how that can be exploited algorithmically to solve the decomposition problem. The existence of an algorithm with time complexity $\mathrm{O}(n^2)$ was previously only known for the permutahedron. Our algorithm not only generalizes this to the scheduling polytope, but also adds a completely new, geometric interpretation. The different parts from Chapter 5 are published as [35, 36] and [37, 38].

Publications underlying this thesis:

[34] R. Hoeksma and M. Uetz. The price of anarchy for minsum related machine scheduling. In R. Solis-Oba and G. Persiano, editors, *Approximation and Online Algorithms*, volume 7164 of *Lecture Notes in Computer Science*, pages 261–273. Springer, 2012.

[35] R. Hoeksma and M. Uetz. Two dimensional optimal mechanism design for a sequencing problem. In M. Goemans and J. Corréa, editors, *Integer Pro-*

gramming and Combinatorial Optimization, volume 7801 of *Lecture Notes in Computer Science*, pages 242–253. Springer, 2013.

[**36**] R. Hoeksma and M. Uetz. Optimal mechanism design for a sequencing problem with two dimensional private data. Invited for publication in Operations Research. Under review, 2014.

[**37**] R. Hoeksma, B. Manthey, and M. Uetz. Decomposition algorithm for the single machine scheduling polytope. In P. Fouilhoux, L. E. N. Gouveia, A. R. Mahjoub, and V. T. Paschos, editors, *Combinatorial Optimization*, Lecture Notes in Computer Science, pages 280–291. Springer, 2014.

[**38**] R. Hoeksma, B. Manthey, and M. Uetz. Decomposition algorithm for the single machine scheduling polytope. Submitted to Discrete Optimization. Under review, 2014.

[**39**] R. Hoeksma, H. Nguyen, and M. Uetz. Fast and scalable mechanism design for a single machine sequencing problem with private data. Manuscript, 2014.

CHAPTER 2

Scheduling selfish jobs on related machines

In this chapter we consider scheduling multiple jobs on multiple machines. We compare equilibrium outcomes to optimal outcomes.

We treat the related machine scheduling problem. We are given n jobs, each with given non-preemptive processing requirements. Each of these jobs needs to be processed on exactly one of m machines, where the machines may have different speeds. The total processing time needed to process a Job j with processing requirement $p_j \in \mathbb{R}^+$ on a Machine i with speed $s_i \in \mathbb{R}^+$ is p_j/s_i. We aim to minimize the sum of (or average over) the completion times of all jobs. We refer to this problem as the minsum related machine scheduling problem. This is one of the classical models in the area of scheduling and was solved, already in the 1960s, by Conway et al. [14]. In the 3-field notation of Graham et al. [27], the problem is denoted $\mathrm{Q} \mid \mid \sum C_j$. The problem is a special case of the, more general, unrelated machine scheduling problem, $\mathrm{R} \mid \mid \sum C_j$, where the processing time of Job j on Machine i can be any value $p_{ij} \in \mathbb{R}^+$.

We look at this problem from a game theoretic angle, where the jobs are selfish players. Each of the jobs selfishly chooses the machine on it will be processed. When, for each machine a sequencing rule for the jobs is given, the game is well defined and we can investigate its Nash equilibria and the price of anarchy. In this chapter we mainly discuss the setting where on each machine the jobs are processed in shortest processing time first (SPT) order. For this setting pure Nash equilibria always exist. While scheduling the jobs in SPT order is optimal on a local, per machine, level, the resulting Nash equilibria in general are not globally optimal. Notice that the problem that we have described so far is an evaluation of SPT as a coordination mechanism as defined by Christodoulou et al. [11], who suggested to use local scheduling rules per machine in order to influence the dynamics of the game and thereby the quality of the corresponding equilibrium outcomes. We will also briefly discuss other coordination mechanisms for this problem.

Our contribution is an analysis of the price of anarchy for the minsum related machine scheduling game. More specifically, our main result is a proof that the price of anarchy is at most 2. Our analysis uses semi-smoothness. Therefore, the results extend beyond pure Nash equilibria. We also give a parametric instance to show that the price of anarchy cannot be less than $e/(e-1) \approx 1.58$. This lower bound even holds for Pure Nash equilibria.

The literature on analysis of the price of anarchy for scheduling problems is very extensive. Most notably, following the work by Koutsoupias and Papadimitriou [45], which introduced the price of anarchy measure, a lot of work was done on the, egalitarian[1], makespan objective, $C_{\max}(= \max_j C_j)$, as social choice function [4, 11, 18, 42, 45, 70]. Like this chapter, recent publications focus more on utilitarian social choice functions [6, 12, 15, 60].

The papers by Correa and Queyranne [15] and Cole et al. [12] are closely related to our work. Both papers address the same problem as we do, but with additional job weights w_j and in the more general context of unrelated machine scheduling. One of the main results in both papers is a proof that the price of anarchy is 4 when machines sequence their jobs locally optimal, that is, according to non-increasing ratios of weight over processing time. Cole et al. [12] also give an instance which establishes a lower bound of 4 for the price of anarchy, even in the unweighted case. The results from this chapter fit nicely into that context.

The organization of this chapter is as follows. In Section 2.1 we briefly recap an algorithm by Horowitz and Sahni [40], which finds optimal solutions for the minsum related machine scheduling problem. We then give a new characterization of such optimal solutions, which is interesting in its own right and crucial for the subsequent analysis. In Section 2.2.1 we prove that the price of anarchy is not greater than 2. We do that by showing that the game is $(2, 0)$-semi-smooth. In this proof, we use the characterization of optimal solutions from Section 2.1. Section 2.2.2 describes a parametric instance, for which we show that its price of anarchy is equal to $e/(e-1) > 1.5819$. In Section 2.4 we compare our outcomes for the SPT coordination mechanism to other coordination mechanisms.

2.1 Characterization of optimal solutions

In this section we first give an algorithm that finds optimal solutions for the minsum related machine scheduling optimization problem, the *Minimum Mean Flow Time* (MFT) algorithm as described by Horowitz and Sahni [40, p. 321]. From that algorithm we establish a new characterization for optimal solutions for minsum related machine scheduling. This characterization is crucial to our analysis in Section 2.2.1.

We denote by N the set of n jobs and by M the set of m machines. Each Job j has a processing requirement p_j and each Machine i has a speed s_i. The time it takes Machine i to process Job j is equal to $p_{ij} = p_j/s_i$. W.l.o.g. we assume that $p_1 \leq p_2 \leq \cdots \leq p_n$ and $s_1 \leq s_2 \leq \cdots \leq s_m$. Ties on the ordering are assumed to be broken consistently and we assume this is done based on index.

The minsum related machine scheduling problem is solved in $O(mn \log n)$ computation time by the MFT algorithm. It is a refinement of the simple matching solution presented earlier by Conway et al. [14, pp. 78-79]. The MFT algorithm

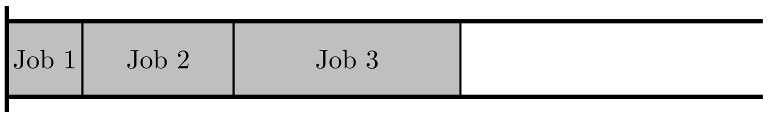
[1]See Myerson [50] for a discussion of utilitarian and egalitarian social choice functions. The interpretation of makespan as egalitarian indeed makes sense in models where the objectives of the job-agents is the total load of the machine they are processed on, as for example in Koutsoupias and Papadimitriou [45].

computes the optimal assignment of jobs to machines by considering them descending in processing requirement, and the jobs eventually assigned to a given machine are then sequenced ascending in their processing time.

For a single machine we know from Theorem 1.2 that scheduling the jobs in order of ascending processing times is optimal for minimizing the sum of completion times. The optimal order of the jobs on a single machine trivially remains the same if we scale all processing times by a factor of s (i.e. the speed of the machine). As an alternate proof of optimality of the SPT rule for the single machine case, the contribution of a job can be measured by its position in the schedule and its processing time. This follows from rewriting the objective function as follows. Let o be an ordering of the jobs and let $o(k)$ denote the k-th job in this ordering, then

$$\sum_{k=1}^{n} C_{o(k)} = \sum_{k=1}^{n}\sum_{l=1}^{k} p_{o(l)} = \sum_{k=1}^{n}(n-k+1)p_{o(k)} \quad .$$

From this we see immediately that any optimal order needs to be an SPT order. See Figure 2.1 for an illustration of this computation.



$$\begin{array}{rrrrcl}
p_1 & & & & = & C_1 \\
p_1 + & p_2 & & & = & C_2 \\
p_1 + & p_2 & + & p_3 & = & C_3 \quad + \\
\hline
3p_1 + & 2p_2 & + & 1p_3 & = & \sum_j C_j
\end{array}$$

Figure 2.1: On a single machine a jobs position determines its total contribution to the sum of completion times objective.

The idea for a single machine, from the proof above, can be extended to the case of related machines. This results in the MFT algorithm [40], Algorithm 2.1.

Algorithm 2.1 MFT Algorithm for minsum related machine scheduling

```
1: For each Machine $i$ set $z_i = 0$
2: while Not all jobs are placed do
3:     Take from the unscheduled jobs the longest Job $j$
4:     Assign Job $j$ to the machine with the smallest value of $(z_i + 1)/s_i$
5:     For that machine update $z_i = z_i + 1$
6: end while
7: Sort the jobs on each machine in SPT order
```

Similar to the single machine case, the different values $(z_i + 1)/s_i$ are the values for a job's possible positions in the schedule, as, in general, the x-th last job on a machine contributes to the objective value x times its processing requirement divided by the machines speed. The algorithm assigns the currently longest unscheduled job to the machine with the currently smallest position value.

Theorem 2.1 (Horowitz and Sahni [40]). *Any optimal schedule for* $\mathrm{Q}\,|\,|\sum C_j$ *can be computed by the MFT algorithm with the proper tie breaking rule.*

From here on we assume that each machine has a predetermined scheduling rule to determine the schedule for the jobs assigned to it. Unless stated otherwise, this is the SPT rule. That said, we will identify a schedule with an n-vector σ, where σ_j is the machine on which Job j is scheduled.

Next, let $z(\sigma, j)$ be the vector such that $z_i(\sigma, j) = |\{k > j | \sigma_k = i\}|$, is the number of jobs on Machine i in schedule σ that have higher index than j. Then, any schedule σ is optimal if and only if [2]

$$\frac{z_{\sigma_j}(\sigma, j) + 1}{s_{\sigma_j}} \le \frac{z_i(\sigma, j) + 1}{s_i} \text{ for all jobs } j \text{ and all machines } i \ . \tag{2.1}$$

This because, for all machines i, $(z_i(\sigma, j) + 1)/s_i$ is the value of the next position on Machine i upon placement of Job j by the MFT algorithm. $p_j(z_{\sigma_j}(\sigma, j) + 1)/s_{\sigma_j}$ is exactly the contribution of Job j to the objective value in schedule σ. The sum of these contributions needs to be minimized by any optimal schedule. The following lemma provides our new characterization of optimal solutions.

Lemma 2.2. *A schedule σ is optimal for* $\mathrm{Q}\,|\,|\sum C_j$ *if and only if* [3]

$$\frac{z_i(\sigma, j) + 1}{s_i} \ge \frac{z_\ell(\sigma, j)}{s_\ell} \textit{ for all machines } i \textit{ and } \ell \ . \tag{2.2}$$

Proof. We show that (2.2) is true if and only if (2.1) is true. Let σ be an optimal schedule and let the ordering of the jobs be fixed and in SPT order. Note that $z_i(\sigma, j) \ge z_i(\sigma, k)$ for all machines i and all jobs $k \ge j$. Therefore, we have from (2.1) that

$$\frac{z_i(\sigma, j) + 1}{s_i} \ge \frac{z_i(\sigma, k) + 1}{s_i} \ge \frac{z_{\sigma_k}(\sigma, k) + 1}{s_{\sigma_k}} \ ,$$

for all machines i and all jobs $k \ge j$. Since for any Machine ℓ either $z_\ell(\sigma, j) = 0$, or there is a Job $k > j$ such that $\sigma_k = \ell$ and $z_\ell(\sigma, j) = z_{\sigma_k}(\sigma, j) = z_{\sigma_k}(\sigma, k) + 1$, it follows that

$$\frac{z_i(\sigma, j) + 1}{s_i} \ge \frac{z_\ell(\sigma, j)}{s_\ell} \text{ for all machines } i \text{ and } \ell \ .$$

[2] In case of ties in the SPT ordering, there exist multiple optimal schedules, produced by interchanging symmetric jobs, jobs with equal processing times, in any optimal schedule. In this case, (2.1) and (2.2) describe optimal schedules that correspond to one particular SPT ordering and the rest can be obtained by interchanging symmetric jobs.

[3] See Footnote 2

Now let σ be a schedule that satisfies (2.2) and suppose it does not satisfy (2.1). Then there exist a Job $j \in N$ and a Machine $i \in M$ such that

$$\frac{z_{\sigma_j}(\sigma, j) + 1}{s_{\sigma_j}} > \frac{z_i(\sigma, j) + 1}{s_i} \; .$$

However, then we have for Job $j-1$ that

$$\frac{z_{\sigma_j}(\sigma, j-1)}{s_{\sigma_j}} = \frac{z_{\sigma_j}(\sigma, j) + 1}{s_{\sigma_j}} > \frac{z_i(\sigma, j) + 1}{s_i} = \frac{z_i(\sigma, j-1) + 1}{s_i} \; ,$$

which contradicts (2.2). □

An intuitive interpretation for (2.2) is that, when applying the MFT algorithm, a job that is placed on a machine can not get a better position than the jobs already placed on a machine. While it is intuitive that this is indeed a necessary condition for the optimal solution, the intuition that it is also sufficient is not that clear. In that sense, it is indeed a nontrivial reformulation of (2.1).

2.2 Price of anarchy for the SPT scheduling rule

In this section we provide both an upper and a lower bound on the price of anarchy for the SPT scheduling rule for the related machine scheduling game.

We denote schedules in the same way as in Section 2.1. With slight abuse of notation, we let σ also represent the strategy profile of the players, such that σ_j is the machine chosen by Job j. Recall that σ_{-j} denotes the $(n-1)$-vector obtained from σ by deleting σ_j, such that $\sigma = (\sigma_j, \sigma_{-j})$. For the problem Q $| \; | \sum C_j$ with SPT as local scheduling rule, a strategy profile $\nu = (\nu_j, \nu_{-j})$ is a pure Nash equilibrium if and only if for all jobs j and all machines i,

$$\sum_{\substack{k \le j \\ \nu_k = \nu_j}} \frac{p_k}{s_{\nu_j}} \le \sum_{\substack{k < j \\ \nu_k = i}} \frac{p_k}{s_i} + \frac{p_j}{s_i} \; . \tag{2.3}$$

It is well known [32] that the *Ibarra-Kim* algorithm [41] constructs all Nash equilibria depending on the way ties are broken. This is even true for the more general unrelated machine scheduling problem [32, 42]. For related machines the algorithm is described in pseudo-code by Algorithm 2.2.

Algorithm 2.2 Ibarra-Kim Algorithm for problem Q $| \; | \sum C_j$

```
1: while Not all jobs are placed do
2:     Take from the unscheduled jobs the shortest Job $k$
3:     Let Machine $l$ be the machine where Job $k$ has minimal completion time
4:     Schedule Job $k$ directly after the jobs already scheduled on Machine $l$
5: end while
```

The Ibarra-Kim algorithm was originally designed as an approximation algorithm for unrelated machine scheduling [41]. Therefore, the main result that we discuss in this chapter is also an analysis of an upper and lower bound to the performance of this, simple, greedy, algorithm, since the outcomes exactly coincide with Nash equilibria. To the best of our knowledge these performance bounds for the related machine scheduling problem $Q \mid \mid \sum C_j$ have not yet been analyzed. Most probably because the problem to find optimal solutions was settled long before by Conway et al. [14].

2.2.1 Upper bound on the price of anarchy

Here we establish an upper bound on the price of anarchy for minsum related machine scheduling. Our proof uses semi-smoothness for cost-minimization games from Definition 1.12. For the proof, we use the characterization of the optimal solution from Lemma 2.2.

In the following, let σ be an optimal schedule resulting from the MFT algorithm and recall that for the objective value in the optimal solution σ we have

$$\sum_{j=1}^{n} C_j(\sigma) = \sum_{j=1}^{n} \left(z_{\sigma_j}(\sigma, j) + 1\right) \frac{p_j}{s_{\sigma_j}} \ .$$

The next theorem is the main result of this chapter.

Theorem 2.3. *The price of anarchy for the minsum related machine scheduling problem, $Q \mid \mid \sum C_j$, with SPT as local scheduling rule is no greater than 2.*

Proof. We show that the game is $(2, 0)$-semi-smooth, by showing that

$$\sum_{j=1}^{n} C_j(\sigma_j, \nu_{-j}) \leq 2 \sum_{j=1}^{n} C_j(\sigma) \ , \tag{2.4}$$

for an optimal schedule σ and any strategy profile ν.

Let $N_i(\sigma) = \{j | \sigma_j = i\}$ be the set of jobs scheduled on Machine i in the optimal solution, σ. Likewise, let $N_i(\nu) = \{j | \nu_j = i\}$ be the set of jobs scheduled on Machine i in schedule ν. For any Job j in $N_i(\sigma)$, its completion time $C_j(\sigma_j, \nu_{-j})$ consists of the processing times of all jobs that are on Machine i in ν and that have smaller index than j, plus its own processing time on Machine i. Summing the completion times of all jobs that are on Machine i in the optimal solution gives us

$$\begin{aligned} \sum_{j \in N_i(\sigma)} C_j(\sigma_j, \nu_{-j}) &= \sum_{j \in N_i(\sigma)} \left(\frac{p_j}{s_i} + \sum_{\substack{k \in N_i(\nu) \\ k<j}} \frac{p_k}{s_i} \right) \\ &= \sum_{j \in N_i(\sigma)} \frac{p_j}{s_i} + \sum_{j \in N_i(\sigma)} \sum_{\substack{k \in N_i(\nu) \\ k<j}} \frac{p_k}{s_i} \ . \end{aligned} \tag{2.5}$$

Note that the number of times that the processing time of Job k is counted on the right hand side of (2.5) equals the number of jobs with higher index than j on Machine i in the optimal solution, times $\frac{1}{s_i}$. In other words, the second part of (2.5) can be rewritten as

$$\sum_{j \in N_i(\sigma)} \sum_{\substack{k \in N_i(\nu) \\ k<j}} \frac{p_k}{s_i} = \sum_{k \in N_i(\nu)} z_i(\sigma, k) \frac{p_k}{s_i} \quad .$$

This gives us

$$\sum_{j \in N_i(\sigma)} C_j(\sigma_j, \nu_{-j}) = \sum_{j \in N_i(\sigma)} \frac{p_j}{s_i} + \sum_{k \in N_i(\nu)} z_i(\sigma, k) \frac{p_k}{s_i} \quad .$$

Now, note that in this expression, by definition, $\sigma_j = \nu_k = i$, so

$$\sum_{j \in N_i(\sigma)} C_j(\sigma_j, \nu_{-j}) = \sum_{j \in N_i(\sigma)} \frac{p_j}{s_{\sigma_j}} + \sum_{k \in N_i(\nu)} z_{\nu_k}(\sigma, k) \frac{p_k}{s_{\nu_k}} \quad .$$

Summing over all machines i leads to

$$\begin{aligned} \sum_{j=1}^{n} C_j(\sigma_j, \nu_{-j}) &= \sum_{i=1}^{m} \sum_{j \in N_i(\sigma)} C_j(\sigma_j, \nu_{-j}) \\ &= \sum_{i=1}^{m} \sum_{j \in N_i(\sigma)} \frac{p_j}{s_{\sigma_j}} + \sum_{i=1}^{m} \sum_{k \in N_i(\nu)} z_{\nu_k}(\sigma, k) \frac{p_k}{s_{\nu_k}} \\ &= \sum_{j=1}^{n} \frac{p_j}{s_{\sigma_j}} + \sum_{j=1}^{n} z^{\sigma}_{\nu_j}(j) \frac{p_j}{s_{\nu_j}} \quad . \end{aligned}$$

From Lemma 2.2 we know that

$$\sum_{j=1}^{n} z^{\sigma}_{\nu_j}(j) \frac{p_j}{s_{\nu_j}} \leq \sum_{j=1}^{n} \left(z_{\sigma_j}(\sigma, j) + 1 \right) \frac{p_j}{s_{\sigma_j}} = \sum_{j=1}^{n} C_j(\sigma) \quad . \tag{2.6}$$

Also, the completion time of any job is at least its processing time on the machine it is scheduled on, so

$$\sum_{j=1}^{n} \frac{p_j}{s_{\sigma_j}} \leq \sum_{j=1}^{n} C_j(\sigma) \quad . \tag{2.7}$$

Combining the above, we get

$$\sum_{j=1}^{n} C_j(\sigma_j, \nu_{-j}) \leq 2 \sum_{j=1}^{n} C_j(\sigma)$$

for all strategy profiles ν. □

2.2.2 Lower bound on the price of anarchy

In this section we describe a parametric instance which has a price of anarchy that is asymptotically equal to $e/(e-1)$. For these instances the Nash equilibrium is the schedule with all jobs on the fastest machine (which is easily shown to be an upper bound on the quality of Nash equilibria in general, so in that sense, this is a worst case scenario).

Instance 2.1. Let $\mathcal{I}$ be the parametric group of instances $I(s)$ that satisfy the following. $I(s)$ has m machines, one of which has speed $s > 1$, while all the other machines have speed 1. Let all speeds be integer. Furthermore, $I(s)$ has $n = m+s-1$ jobs, with processing requirement equal to

$$p_j = \begin{cases} 1 & \text{if } 1 \leq j \leq s \\ x^{j-s} & \text{if } s+1 \leq j \leq n \end{cases} ,$$

where $x = s/(s-1)$.

Lemma 2.4. *Instances from $\mathcal{I}$ have a Nash equilibrium with all jobs on the fastest machine.*

Proof. In the schedule with all jobs in SPT order on the fastest machine, the completion time of a Job $j < s$ is equal to

$$C_j = \sum_{k=1}^{j} \frac{p_k}{s} = \sum_{k=1}^{j} \frac{1}{s} = \frac{j}{s} \leq 1 \ . \tag{2.8}$$

For a Job $j \geq s$, the completion time is equal to

$$\begin{aligned} C_j = \sum_{k=1}^{j} \frac{p_k}{s} &= \frac{s-1}{s} + \sum_{k=s}^{j} \frac{\left(\frac{s}{s-1}\right)^{k-s}}{s} \\ &= \frac{1}{s}\left(s-1+\sum_{k=0}^{j-s}\left(\frac{s}{s-1}\right)^{k}\right) \\ &= \frac{1}{s}\left(s-1+\frac{\left(\frac{s}{s-1}\right)^{j-s+1}-1}{\left(\frac{s}{s-1}\right)-1}\right) \\ &= \frac{1}{s}\left(s-1+(s-1)\left(\frac{s}{s-1}\right)^{j-s+1}-(s-1)\right) \\ &= \left(\frac{s}{s-1}\right)^{j-s} = p_j \ . \end{aligned} \tag{2.9}$$

Thus, since all other machines have speed 1, the Nash equilibrium condition, (2.3), holds. □

We use Instance 2.1 to compute the lower bound of $e/(e-1)$ on the price of anarchy.

Theorem 2.5. *The price of anarchy for the minsum related machine scheduling problem,* $\mathrm{Q}\,|\,|\sum C_j$, *with SPT local scheduling rule, is no less than* $e/(e-1) \approx 1.58$.

Proof. Consider instances $I(s)$ from $\mathcal{I}$ as defined above. In the optimal solution the s longest jobs are on the fastest machine. All other jobs are on a slow machine. So the objective value in the optimal solution is equal to

$$\begin{aligned}
\mathrm{OPT}(I(s)) &= \sum_{j=1}^{s-1} p_j + \sum_{j=s}^{n-s} p_j + \sum_{j=n-s+1}^{n} \sum_{k=n-s+1}^{j} \frac{p_k}{s} \\
&= \sum_{j=1}^{s-1} 1 + \sum_{j=s}^{n-s} x^{j-s} + \sum_{j=n-s+1}^{n} \sum_{k=n-s+1}^{j} \frac{x^{k-s}}{s} \\
&= s-1+\sum_{j=0}^{n-2s} x^j + \sum_{j=n-s+1}^{n} \frac{1}{s}\left(\sum_{k=0}^{j-s} x^k - \sum_{k=0}^{n-2s} x^k\right) \\
&= s-1+(s-1)x^{n-2s+1}-(s-1)+\sum_{j=n-s+1}^{n}\left(x^{j-s}-x^{n-2s}\right) \\
&= (s-1)x^{n-2s+1}+\sum_{j=n-2s+1}^{n-s} x^j - \sum_{j=n-s+1}^{n} x^{n-2s} \\
&= (s-1)x^{n-2s+1}+(s-1)x^{n-s+1}-(s-1)x^{n-2s+1}-sx^{n-2s} \\
&= (s-1)x^{n-s+1}-(s-1)x^{n-2s+1}\quad . \qquad (2.10)
\end{aligned}$$

From Lemma 2.4 we know that the schedule with all jobs on the fastest machine is a Nash equilibrium. From (2.8) and (2.9) we know that the completion time of the jobs in this schedule is equal to

$$C_j = \begin{cases} \frac{j}{s} & \text{if } j \le s-1 \\ \left(\frac{s}{s-1}\right)^{j-s} & \text{otherwise} \end{cases} \quad .$$

From this we compute the objective value in the Nash equilibrium

$$\begin{aligned}
\mathrm{NE}(I(s)) &= \sum_{j=1}^{s-1} \frac{j}{s} + \sum_{j=s}^{n} x^{j-s} \\
&= \frac{s(s-1)}{2s} + \sum_{j=0}^{n-s} x^{j} \\
&= \frac{(s-1)}{2} + (s-1)x^{n-s+1} - (s-1) \\
&= (s-1)x^{n-s+1} - \frac{(s-1)}{2} \quad . \qquad (2.11)
\end{aligned}$$

Combining (2.10) and (2.11) gives us the price of anarchy:

$$\begin{aligned}
\mathrm{POA}(I(s)) &= \frac{(s-1)x^{n-s+1} - \frac{(s-1)}{2}}{(s-1)x^{n-s+1} - (s-1)x^{n-2s+1}} \\
&= \frac{x^{n-s+1} - \frac{1}{2}}{x^{n-s+1} - x^{n-2s+1}} \\
&= \frac{x^{s} - \frac{1}{2}x^{-(n-2s+1)}}{x^{s} - 1} \\
&= \frac{\left(\frac{s}{s-1}\right)^{s} - \frac{1}{2}\left(\frac{s}{s-1}\right)^{-(n-2s+1)}}{\left(\frac{s}{s-1}\right)^{s} - 1} \quad . \qquad (2.12)
\end{aligned}$$

Now, if we let n go to infinity, (2.12) becomes:

$$\lim_{n\to\infty} \mathrm{POA}(I(s)) = \frac{\left(\frac{s}{s-1}\right)^{s}}{\left(\frac{s}{s-1}\right)^{s} - 1} \quad , \qquad (2.13)$$

and letting s also go to infinity, (2.13) goes to $e/(e-1) \approx 1.58$. □

2.3 Special cases

Two special cases of this problem arise when either the machines or the jobs are all identical. In both these cases all pure Nash equilibria are optimal solutions. However, even if both the machines and the jobs are identical, i.e. $s_i = 1$ for all i and $p_j = 1$ for all j, mixed Nash equilibria have price of anarchy equal to $3/2$.

Theorem 2.6. *The problem of scheduling n identical jobs on $m = n$ identical machines has a mixed Nash equilibrium with sum of completion times equal to $3/2 - 1/(2m)$ times the sum of completion times in the optimal solution.*

Proof. The optimal solution to this problem schedules each job on a machine by itself and has sum of completion times equal to n. Recall that ties are broken consistently over all machines and according to the index of the jobs. Now consider the mixed Nash equilibrium, ν, where each job is scheduled on any machine with probability $1/n$. Then for each Job j, the load of the jobs with index less than j is divided equally over all machines. So Job j can not improve, since all machines appear the same to it. Therefore, ν is indeed a mixed Nash equilibrium. Now, for any Job j, the expected completion time is equal to

$$\mathbb{E}_{\sigma\sim\nu}C_j(\sigma) = 1 + \frac{j-1}{n} \ .$$

So summing over all jobs gives us

$$\sum_{j=1}^{n}\mathbb{E}_{\sigma\sim\nu}C_j(\sigma) = \sum_{j=1}^{n} 1 + \frac{j-1}{n} = n + \frac{n(n-1)}{2}n = \frac{3n}{2} - \frac{1}{2} \ .$$

Since $n = m$, dividing by n gives $3/2 - 1/(2m)$. □

The identical machine model, where all machines have speed 1, has robust price of anarchy of exactly $3/2 - 1/(2m)$. This result was also found by Rivera Letelier [61] and Rahn and Schäfer [60]. Here we give a short and simple proof that we discussed in private communication with J.R. Correa. The proof follows the framework of β-niceness as defined by Augustine et al. [3] and, its extended version, (λ, μ)-niceness as defined by Anshelevich et al. [2]. In fact, we prove that the game is $(\frac{3}{2} - \frac{1}{2m})$-nice and thus also $(\frac{3}{2} - \frac{1}{2m}, 0)$-nice. This implies bounds on Nash equilibria, mixed Nash equilibria and correlated equilibria, but not coarse correlated equilibria [2].

Theorem 2.7. *The price of anarchy for the minsum identical machine scheduling problem, that schedules the jobs in a fixed order on all machines, is* $3/2 - 1/(2m)$.

Proof. Let the jobs be indexed according to the order in which the machines process them. For any strategy profile ν and any Job j, let ν'_j be a best response of j to ν_{-j}. Thus ν'_j is any one strategy that minimizes Job j's completion time, given that all other jobs keep their original strategy.

$$C_j(\nu'_j, \nu_{-j}) \le C_j(\nu^*_j, \nu_{-j}) \qquad \text{for all } j \in N,\ \nu \in M^n \text{ and } \nu^* \in M.$$

Since all machines have speed 1, minimizing completion time is the same as minimizing start time. Therefore, the completion time for Job j in profile (ν'_j, ν_{-j}) can never be greater than the situation in which all machines have equal load from the jobs up to Job j. In that situation the load of each machine would be exactly $\sum_{k=1}^{j-1} p_k/m$. Since this holds for all jobs $j \in N$, we have

$$\sum_{j\in J} C_j(\nu'_j, \nu_{-j}) \le \sum_{j\in J}\sum_{k=1}^{j-1}\frac{p_k}{m} + \sum_{j\in J} p_j \ .$$

Let σ be a strategy profile that results in an optimal solution. The lower bound on the optimal solution from Eastman et al. [21, Thm. 1] gives:

$$\sum_{j\in J}\sum_{k=1}^{j}\frac{p_k}{m}+\left(\frac{1}{2}-\frac{1}{2m}\right)\sum_{j\in J}p_j \le \sum_{j\in J}C_j(\sigma) \ .$$

Therefore

$$\begin{aligned}
\sum_{j\in J}C_j(\nu'_j,\nu_{-j}) &\le \sum_{j\in J}\sum_{k=1}^{j-1}\frac{p_k}{m}+\sum_{j\in J}p_j && (2.14)\\
&\le \sum_{j\in J}\sum_{k=1}^{j}\frac{p_k}{m}+\left(1-\frac{1}{m}\right)\sum_{j\in J}p_j && (2.15)\\
&\le \sum_{j\in J}C_j(\sigma)+\left(\frac{1}{2}-\frac{1}{2m}\right)\sum_{j\in J}p_j && (2.16)\\
&\le \left(\frac{3}{2}-\frac{1}{2m}\right)\sum_{j\in J}C_j(\sigma) \ . && (2.17)
\end{aligned}$$

This proves that the game is $(\frac{3}{2}-\frac{1}{2m},0)$-nice and, thus, that the price of anarchy is equal to $\frac{3}{2}-\frac{1}{2m}$. □

2.4 Other scheduling rules

A natural question to ask is if there exist other scheduling rules that outperform the SPT scheduling rule. In fact, it is not hard to see that for any instance there exists a set of scheduling rules that results in optimal solutions.

Theorem 2.8. *For any instance of the related machine scheduling game, there exists a set of scheduling rules for the machines such that any Nash equilibrium is an optimal solution.*

Proof. Let σ be an optimal assignment of the jobs to the machines. Let N_i be the set of jobs that are scheduled on Machine i in σ. Let $\overline{N}_i$ be the set of jobs that are not scheduled on Machine i in σ. Consider the following set of scheduling rules: for any Machine i jobs in N_i are scheduled before jobs not in N_i. Jobs within N_i and $\bar{N}_i$ are scheduled in optimal order, which in this case is SPT order. Now let ν be a Nash equilibrium assignment. Then $C_j(\nu) \le C_j(\sigma_j,\nu_{-j})$ for all jobs j and since only jobs from N_{σ_j} could be scheduled before Job j on machine σ_j, $C_j(\sigma) \ge C_j(\sigma_j,\nu_{-j}) \ge C_j(\nu)$ for all jobs j. Therefore, ν must be an optimal schedule. □

Note that the proof above uses a $(1,0)$-smoothness argument, which directly implies that the robust price of anarchy for this game is equal to 1. Moreover it holds for any machine scheduling settings, where jobs only care about their completion time and the objective function is monotone in those completion times. In other words, if none of the jobs completion times increase neither does the objective function.

While Theorem 2.8 shows that, for any given instance, we can find a set of local scheduling rules, such that Nash equilibria are optimal solutions, these heavily rely on the fact that we know the complete instance in advance. It seems reasonable to assume that we have to decide on these scheduling rules without knowing which jobs we will encounter. Christodoulou et al. [11] introduce *coordination mechanisms* as a model that fits exactly this idea. They define a coordination mechanism for a scheduling problem simply as a set of local scheduling rules. They note specifically that "the scheduling policies should be defined before the set of loads." Therefore, a coordination mechanism defines scheduling rules independent of the set of all jobs. Recall that the single machine scheduling polytope for half times, (1.2) and (1.3), describes the convex hull of all feasible start time vectors without idle time. In general we want to allow coordination mechanisms to schedule the jobs with idle time. Therefore, we consider the, unbounded, scheduling polyhedron for completion times, described by:

$$\sum_{j\in K} C_j p_j \geq g(K) - \frac{1}{2}\sum_{j\in K} p_j^2 \qquad \text{for all } K \subset N \ . \tag{2.18}$$

This includes not only schedules with idle time, but also allows preemption and randomization. For a given processing tmes vector p let $Q^C(p)$ denote the scheduling polyhedron for completion times. We define a coordination mechanism for $Q||\sum C_j$ as follows.

Definition 2.1 (Coordination mechanism for $Q||\sum C_j$)**.** A coordination mechanism defines, given a set of machines, M, for each Machine i a function $f_i : \mathbb{R}^k_+ \to \mathbb{R}^k_+$, for all positive integers k. That is, for any set of k jobs, represented by a vector of processing times, p, the function $f_i(p)$ is a vector of completion times contained in the scheduling polyhedron, $Q^C(p)$.

As we have defined coordination mechanisms, above, it fits the definition of strongly local coordination mechanisms by Azar et al. [4]. They define this concept, for unrelated machines, as coordination mechanisms in which the ordering of jobs on Machine i only depends on the processing times of the jobs on Machine i. Our definition of coordination mechanisms can be naturally extended for these more general settings, where jobs have more properties than only processing times, for example weights, release dates or different processing times on each machine. We make this extension, simply by allowing each Machine i to define a function, $f_i : T^k_+ \to \mathbb{R}^k_+$, where T denotes the set of all possible types that a job can have. These types, then, include all the parameters that define such a job.

The following theorem shows that the coordination mechanism setting indeed rules out trivial solutions, such as the one in Theorem 2.8.

Theorem 2.9. *No coordination mechanism for $Q||\sum C_j$ has pure price of anarchy equal to 1.*

Proof. Consider an instance with two machines. Machine 1 has speed $s_1 = 2$ and Machine 2 has speed $s_2 = 3$. First consider a job set $N = \{1, 2\}$, with processing requirements $p_1 = 1$ and $p_2 = 2$. The unique optimal solution is to schedule Job 1 on Machine 1 and Job 2 on Machine 2, with sum of completion times equal to 7/6. Suppose there is a coordination mechanism for $Q||\sum C_j$ that has pure price of anarchy equal to 1. Then, if Machine 2 gets assigned job set $\{1, 2\}$, and if it would schedule these jobs in SPT order without idle time, then Job 1 achieves a completion time less than 1/2 on Machine 2. Hence, it can never be equilibrium for that job to select Machine 1 and the coordination mechanism fails to compute the optimal solution if the job set is $\{1, 2\}$. Thus, if Machine 2 gets assigned job set $\{1, 2\}$, it can not schedule them, in SPT order without idle time. Now, by introducing yet another job, Job 3, identical to Job 1, we know, by the same reasoning, that Machine 2 cannot use SPT for job sets $\{1, 2\}$ and $\{1, 3\}$. So, the coordination mechanism must fail to compute the optimal solution for job set $\{1, 2, 3\}$. □

Notice that in the above proof, and in Definition 2.1, it is still feasible for a local scheduling rule to explicitly exploit the information about any other machines. In particular, the fact that there are other machines. Otherwise, the above proof would even be simpler, as any machine would need to schedule any job set optimally for that machine, i.e. in SPT order. In other words, the functions f_i, that describe the local scheduling rules, may, in general, be dependent on the vector of machine speeds, and in particular on the number of machines. If that is not allowed, either, the above instance with job set $\{1, 2\}$ even leads to an optimality gap, 8/7. We do not go into further detail on either case here, but rather go back to SPT, or more generally, any coordination mechanism that schedules the jobs according to priority list on each machine, and more specifically, the same priority list on all machines.

Definition 2.2. We call a coordination mechanism a *list coordination mechanism* if there is a single order of the jobs, such that each machine processes the jobs in that order.

If we restrict ourselves to these list coordination mechanisms we can directly see from the proof for Theorem 2.6 that the same reasoning holds and, thus, we get:

Theorem 2.10. *Any list coordination mechanism for the minsum identical machine scheduling problem has mixed price of anarchy at least* $3/2 - 1/(2m)$.

One specific coordination mechanism of interest is a preemptive one where each machine schedules the jobs in SPT order but releases each job only after holding it for a time period that corresponds to the costs it imposes on other jobs on that machine. Cole et al. [12] refer to this coordination mechanism as proportional sharing. They show that, for the unrelated machine model with weights, the price of anarchy of proportional sharing is $1 + \phi$, where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.618$ is the golden ratio. For that model this is significantly better than the price of anarchy for the WSPT rule, which is equal to 4. Rahn and Schäfer [60] find the same coordination mechanism in relation to α-altruistic extensions of scheduling games. These are extensions where the perceived utility of players is partly influenced by the total social costs. Also, this coordination mechanism is closely related to VCG payments for mechanism design, where players compensate each other for the costs they impose by participating in the game, (see e.g. [20, Chap. 15]).

For the related machine scheduling model with minsum objective, the proportional sharing rule results in the following completion times for the players, given a strategy vector σ.

$$C_j(\sigma) = \sum_{\substack{k \in N_{\sigma_j} \\ k \leq j}} p_k + \sum_{\substack{k \in N_{\sigma_j} \\ k > j}} p_k = \sum_{k \in N_{\sigma_j}} \min\{p_k, p_j\} \enspace .$$

While the proportional sharing rule improves the price of anarchy for $R||\sum w_j C_N$, compared to WSPT, the following simple observation shows that, for the proportional sharing rule, we have a lower bound of $2 - \frac{2}{n+1}$ on the price of anarchy for $Q||\sum C_j$.

Consider single machine scheduling problem, with n jobs, with unit processing time and unit weight, $1|p_j = 1|\sum C_j$. The optimal solution is processing the jobs non-preemptively in any order. This has objective value $\frac{n(n+1)}{2}$. With the proportional sharing rule, each Job j has $C_j = n$. Therefore, the objective value is equal to n^2 and the price of anarchy is $2 - \frac{2}{n+1}$.

We see, from the above observation, that even for a single machine, the proportional sharing coordination mechanism does not achieve anything better than a solution that is a factor of $2 - \frac{2}{n+1}$ away from the optimum. Since this holds per machine, it must also hold for $Q||\sum C_j$. Therefore, if we want to find a coordination mechanism that has a price of anarchy better than 2, the upper bound we prove for SPT, the proportional sharing rule is not a suitable candidate.

CHAPTER 3

Scheduling jobs with private information

In this chapter, we address the optimal scheduling mechanism design problem for the single machine scheduling model introduced by Heydenreich et al. [33]. The model is an abstraction of a simple queueing problem with private data: A number of n clients are queueing for a service, the service provider needs to compensate all clients for their waiting time, but waiting costs and service times are private to the clients. It models economic situations where clients queue for a single scarce resource, e.g. scheduling aircraft landing, where multiple planes have to be scheduled to land on one runway [5, 7]. At the same time, the problem is the private information version of one of the most basic and classical machine scheduling models, namely to minimize the total weighted completion time of a given number of non-preemptive jobs with weights w_j and processing times p_j on a single machine. This problem is close to trivial from the optimization point of view, as the optimal sequence is to process the jobs in order of non-increasing ratios w_j/p_j [66]. Once the data w_j and p_j is private, however, the solution is far from trivial, as we will see.

While Heydenreich et al. [33] mainly address the version with single-dimensional private data, we here focus on the case with two-dimensional private data. Indeed, starting with the seminal paper by Myerson [49], optimal mechanism design with single-dimensional private data is pretty well understood, also from an algorithmic point of view [e.g. 31], while algorithmic results for optimal mechanism design with multi-dimensional private data have been obtained only recently [e.g. 1, 8]. Our starting point is the open problem formulated by Heydenreich et al. [33], who ‘leave it as an open problem to identify (closed formulae for) optimal mechanisms for the 2-d case.’ Here, the ‘2-d case’ refers to the problem of computing a Bayes-Nash optimal mechanism for the following sequencing problem on a single machine: There are n jobs with two-dimensional private data, namely a cost per unit time w_j and a processing time p_j. Jobs need to be processed sequentially and non-preemptively, and each job requires a compensation for the disutility of waiting. With given priors on the private data of jobs, the optimal mechanism seeks to minimize the total expected payments made to the jobs, while being BNIC. We answer the open problem formulated by Heydenreich et al. [33], by giving an optimal mechanism and showing that it can be computed in polynomial time. Our solution is based on linear programming techniques and results in an optimal randomized mechanism. In that sense, we do not obtain analytic ‘closed formulae’ for the solution, and our results can be seen in the tradition of ‘automated mechanism design’, as proposed by, among others, Conitzer and Sandholm [13] and Sandholm [64], in that the design of the mechanism itself is based on (integer) linear programming.

The technical contribution of this chapter is the compactification of an exponential size linear programming formulation of the mechanism design problem. This crucial ingredient allows a polynomial time algorithm to compute payments and a so-called interim schedule by solving a polynomial size linear program.

Finally, again in the flavor of automated mechanism design, we present computational results based on the (integer) linear programming formulations. These computations have the primary goal to test and validate hypotheses on the structure of solutions. Our computations, based on randomly generated instances, show that optimal mechanisms in the two-dimensional setting do *not* share several of the nice properties of the solutions to the single-dimensional problem: The scheduling rules of optimal Bayes-Nash incentive compatible mechanisms are not necessarily IIA (a desirable property to be defined later), and neither do optimal Bayes-Nash mechanisms allow an implementation in dominant strategies. This in contrast to the single-dimensional problem which does have these properties [33, 19].

Closely related to our work is a result by Cai et al. [8]. They describe a general framework for mechanism design problems for which they show that an FPTAS exists that finds (near) optimal mechanisms. However the problem described here does not fit directly into their framework, because the model considered here has informational externalities[1]. Still, with some mild adaptations, one can see that the techniques of Cai et al. [8] can be applied also to the problem studied here and even imply that an optimal mechanism can be found in polynomial time. However, these techniques do not lead to an explicit, polynomial size LP model for the optimal mechanism design problem, which we do provide here.

3.1 Definitions, preliminary & related results

We consider a single machine scheduling problem with n agents, denoted by $j \in N$, each owning a job with weight w_j and processing time p_j. We identify jobs with agents. The jobs need to be sequenced (processed) non-preemptively on a single machine, with the interpretation that w_j is Job j's individual cost for waiting one unit of time, while p_j is the time it requires to process Job j. In a schedule that yields a start time S_j for Job j, the cost for waiting is $w_j S_j$. The *type* of a Job j is the two-dimensional vector of weight and processing time, denoted $t_j = (w_j, p_j)$. With t_j being public, the total waiting cost is well known to be minimized by sequencing the jobs in order of non-increasing ratios w_j/p_j, also known as Smith's rule [66].

In the setting we consider here, weight and processing time are private to the agent that owns the job. There is, however, a public belief about this private infor-

[1]That is, the valuation of an agent for a given solution depends on types of other agents, too. If we think of the mechanism design problem as the problem to assign n jobs to positions $1, 2, \ldots, n$, indeed, the valuation that a job has for a given position k depends on the processing times of the jobs on positions $1, \ldots, k-1$.

mation, which is[2]

- the types that Job j might have are $T_j = \{t_j^1, \dots, t_j^{m_j}\}$, and
- the probability of Job j having type t_j^i is $\varphi_j(t_j^i)$, $i = 1 \dots, m_j$.

By $T = T_1 \times \dots \times T_n$ we denote the type space of all jobs, with $t = (t_1, \dots, t_n) \in T$. Define $m := \sum_{j \in N} m_j$, and note that $m \geq n$. For a type $t_j^i \in T_j$, we let w_j^i and p_j^i be the corresponding weight and processing time, respectively. We sometimes abuse notation by identifying i with t_j^i, to avoid excessive notation. Moreover, (t_j, t_{-j}) denotes a type vector where t_j is the type of Job j and t_{-j} are the types of all jobs except j, with $t_{-j} \in T_{-j} := \prod_{k \neq j} T_k$, the set of type vectors excluding j. For given $t \in T$ and $K \subseteq N$, we also define the shorthand notation $\varphi_K(t_K) := \prod_{k \in K} \varphi_k(t_k)$ for the product distribution of the types of jobs in K, particularly $\varphi_{-j}(t_{-j}) := \prod_{k \neq j} \varphi_k(t_k)$.

We assume, just like Heydenreich et al. [33], that the mechanism designer needs to compensate the jobs for waiting by a payment π_j. We seek to compute and implement a (direct) mechanism, consisting of a scheduling rule and a payment rule. More specifically, the mechanism assigns to any type vector $t \in T$ a vector $S(t)$ that represents the start times of all jobs in the sequence selected by the mechanism, together with a vector of compensation payments $\pi(t)$, one for every job. In the mechanism design and auction literature, for obvious reasons, what is a scheduling rule here is referred to as *allocation rule*. Clearly, jobs may have an incentive to strategically misreport their true types in order to receive higher compensation payments. The optimal mechanism that we seek, however, is one that minimizes the total payments made to the jobs. Again like Heydenreich et al. [33], we assume that only larger than the true processing times can be reported by any job, since reporting a processing time smaller than the true processing time is verifiable while processing a job and would leave the job uncompleted.

Myerson's revelation principle [49] makes this problem, like many others [68], amendable to optimization techniques. We considered Bayes-Nash incentive compatible mechanisms. We introduce ES_j^i and π_j^i as shorthand notation for the expected start time and the payment for Job j when he reports to be of type t_j^i. Then a mechanism is BNIC if it fulfills the following, linear constraint

$$\pi_j^i - w_j^i ES_j^i \geq \pi_j^{i'} - w_j^i ES_j^{i'} \quad \text{for all jobs } j \text{ and types } t_j^i, t_j^{i'} \in T_j \ .$$

The expectation ES_j^i is taken over all (truthful) reports of other jobs $t_{-j} \in T_{-j}$. Then, assuming utilities are quasi-linear, the expected utility for Job j with true type t_j^i is $\pi_j^i - w_j^i ES_j^i$ for reporting truthfully, while a false report $t_j^{i'}$ yields expected utility $\pi_j^{i'} - w_j^i ES_j^{i'}$.

[2]Note that the discrete type space makes the problem amendable for (I)LP techniques. Indeed, in the words of Vohra [68], 'nothing of qualitative significance is lost in moving from a continuous to a discrete type space'.

Moreover the mechanism is individually rational, which ensures that the agents are compensated for their waiting time if they report truthfully,

$$\pi_j^i - w_j^i ES_j^i \geq 0 \quad \text{for all jobs } j \text{ and types } t_j^i \in T_j \ .$$

It is interesting to ask if a scheduling rule (more generally, allocation rule) can even be implemented in the stronger *dominant strategy* equilibrium; Manelli and Vincent [47] indeed show the equivalence of BNIC and DSIC implementations for the case of standard single unit private value auctions. In a dominant strategy equilibrium, reporting the true type maximizes the utility of a job not only in expectation but for *any* report t_{-j} of the other jobs. The latter obviously implies the former, but generally not vice versa [26].

In the setting considered here, a mechanism is Bayes-Nash implementable if and only if the expected start times ES_j^i are monotonically increasing in the reported weight w_j^i. The same result holds for dominant strategy implementability. Then the start times, $S_j(t_j^i, t_{-j})$, need to be monotonically increasing in the reported weight, w_j^i, for all $t_{-j} \in T_{-j}$. This is a standard result in single-dimensional mechanism design, see for instance the introductory text by Nisan [53], but it is also true for the two-dimensional problem considered here [33, 19].

For the single-dimensional mechanism design problem, where only weights w_j are private information and processing times p_j are known, the optimal mechanism has a simple structure: It is Smith's rule, but with respect to virtual instead of original weights w_j; see Heydenreich et al. [33] and Duives et al. [19] for details. In this case the optimal Bayes-Nash incentive compatible mechanism can be computed and implemented in polynomial time, and it can even be implemented with the same expected cost in dominant strategies [19].

The problem to find and analyze an optimal mechanism for the two-dimensional optimal mechanism design problem was left open by Heydenreich et al. [33] and Duives et al. [19].

3.2 Problem formulations & linear relaxation

Let us start by giving a natural, albeit exponential size ILP formulation for the mechanism design problem at hand. Recall that $S_j(t)$ denotes the start time of Job j in the sequence selected by the mechanism for given type vector t. Let us denote by σ some sequence of the jobs. We use the natural variables

$$x(\sigma, t) = \begin{cases} 1 & \text{if for type vector } t \text{ sequence } \sigma \text{ is used}\,, \\ 0 & \text{otherwise} \ . \end{cases}$$

Let us denote by $S(\sigma, t)$ the vector of start times of jobs that correspond to type vector t and sequence σ. Note that the parameters $S(\sigma, t)$ can be computed directly from σ and t. Then the formulation reads as follows.

Linear program 1

$$\min \sum_{j \in N} \sum_{i \in T_j} \varphi_j^i \pi_j^i \tag{3.1}$$

$$\pi_j^i \geq w_j^i ES_j^i \qquad \forall j \in J, i \in T_j \tag{3.2}$$

$$\pi_j^i \geq \pi_j^{i'} - w_j^i (ES_j^{i'} - ES_j^i) \qquad \forall j \in N, i, i' \in T_j, p_j^{i'} \geq p_j^i \tag{3.3}$$

$$ES_j^i = \sum_{t_{-j} \in T_{-j}} \varphi(t_{-j}) \sum_{\sigma} x(\sigma, (t_j^i, t_{-j})) S_j(\sigma, (t_j^i, t_{-j})) \qquad \forall j \in N, t_j^i \in T_j \tag{3.4}$$

$$\sum_{\sigma} x(\sigma, t) = 1 \qquad \forall t \in T \tag{3.5}$$

$$x(\sigma, t) \in \{0, 1\} \qquad \forall \sigma \in \Sigma, t \in T \ . \tag{3.6}$$

Here we use the shorthand notation φ_j^i for $\varphi_j(t_j^i)$ and $i \in T_j$ for $t_j^i \in T_j$. Σ denotes the set of all sequences of the jobs in N. The objective (3.1) is the total expected payment. Constraints (3.2) and (3.3) are the individual rationality and incentive compatibility constraints: (3.2) requires the expected payment to at least match the expected cost of waiting when the type is t_j^i, and (3.3) makes sure that the expected utility is maximized when reporting truthfully. The values ES_j^i are also referred to as an *interim schedule*. Indeed, ES_j^i is the expected start time of Job j given it has type i. Here, the expectation is over all types of jobs other than j and equations (3.4) are the feasibility constraints for interim schedules, expressing the fact that the expected start times in the interim schedule need to comply with the scheduling rule encoded by x. While the input size of the mechanism design problem is $\mathrm{O}(m)$, this ILP formulation is colossal as the number of variables $x_\sigma(t)$ equals $|T|\, n!$ with $|T| = \prod_j m_j$, and therefore is potentially doubly exponential.

Observe that, for a given type vector t, $S(\sigma, t)$ are start time vectors of the jobs. These correspond to vertices of the scheduling polytope. Introducing $S = (S_1, \ldots, S_n)$ as variables for the start times of jobs, recall from Theorem 1.1 that the scheduling polytope is defined by

$$\sum_{j \in K} p_j(t) S_j \geq \frac{1}{2} \left(\sum_{j \in K} p_j(t) \right)^2 - \frac{1}{2} \sum_{j \in K} p_j(t)^2 \qquad \forall K \subset N \tag{3.7}$$

$$\sum_{j \in N} p_j(t) S_j = \frac{1}{2} \left(\sum_{j \in N} p_j(t) \right)^2 - \frac{1}{2} \sum_{j \in N} p_j(t)^2 \ , \tag{3.8}$$

where we use $p_j(t)$ to denote the processing time of Job j in type profile t. Observe that (3.8) excludes schedules with idle time. The vertices of the scheduling polytope are exactly all permutation schedules, and hence, any point, S, that satisfies (3.7) and (3.8) represents feasible expected start times of a randomization over (at most n) schedules. It is well known that the scheduling polytope is a polymatroid, which is easily verified via a variable transform to $p(t)S$. Both optimization and separation for the scheduling polytope can be done in time $\mathrm{O}(n^2)$ [22, 58].

3.2.1 Extended formulation in linear ordering variables

It turns out to be convenient for our purpose to consider an extended formulation for the scheduling polytope, namely using linear ordering variables d_{kj},

$$d_{kj}(t) = \begin{cases} 1 & \text{if for type vector } t \text{ we use a schedule where Job } k \text{ precedes Job } j\ , \\ 0 & \text{otherwise}\ . \end{cases}$$

In terms of these variables, for any given vector of types t, start times of jobs are then

$$S_j(t) = \sum_{k \in N} d_{kj}(t) p_k(t)\ .$$

Using linear ordering variables yields the following, extended formulation of the optimal mechanism design problem.

Linear program 2

$$\min \sum_{j \in N} \sum_{i \in T_j} \varphi_j^i \pi_j^i \tag{3.9}$$

$$\pi_j^i \geq w_j^i ES_j^i \qquad \forall j \in N, i \in T_j \tag{3.10}$$

$$\pi_j^i \geq \pi_j^{i'} - w_j^i (ES_j^{i'} - ES_j^i) \qquad \forall j \in N, i, i' \in T_j,\ p_j^{i'} \geq p_j^i \tag{3.11}$$

$$ES_j^i = \sum_{t_{-j} \in T_{-j}} \varphi(t_{-j}) S_j(t_j^i, t_{-j}) \qquad \forall j \in N, i \in T_j \tag{3.12}$$

$$S_j(t) = \sum_{k \in N} d_{kj}(t) p_k(t) \qquad \forall j \in N, t \in T \tag{3.13}$$

$$d_{jj}(t) = 0 \qquad \forall j \in N, t \in T \tag{3.14}$$

$$d_{kj}(t) + d_{jk}(t) = 1 \qquad \forall j, k \in N, j \neq k, t \in T \tag{3.15}$$

$$d_{jk}(t) + d_{kl}(t) \leq 1 + d_{jl}(t) \qquad \forall j, k, l \in N, t \in T \tag{3.16}$$

$$d_{jk}(t) \in \{0, 1\} \qquad \forall j, k \in N, t \in T\ . \tag{3.17}$$

Observe that, in contrast to the previous formulation, the number of variables $d_{jk}(t)$ now equals $n^2 \cdot |T|$. However this formulation is in general exponential as well, since the type space T can be exponential in m, the total number of types of all jobs.

Vertices of the scheduling polytope are the solutions $S(t)$ of (3.13)-(3.17), and moreover the following lemma holds; see for instance Queyranne and Schulz [59, Thm. 4.1]

Lemma 3.1. *A vector $S(t) \in \mathbb{R}$ is contained in the scheduling polytope if and only if there are linear ordering variables d that satisfy* (3.13), (3.15) *and*

$$d_{kj}(t) \in [0, 1] \qquad \forall j, k \in N, t \in T\ . \tag{3.18}$$

Proof. Since any vertex of the scheduling polytope can be described by linear ordering variables that satisfy (3.15), (3.13) and (3.17), it can therefore also be described by linear ordering variables that satisfy (3.13), (3.15) and (3.18). Since any point $S(t)$ in the scheduling polytope is, by definition, a convex combination of such vertices, it can also be described by linear ordering variables that satisfy (3.15), (3.13) and (3.18).

Now let $S(t)$ be described by linear ordering variables that satisfy (3.15), (3.13) and (3.18). Then for any $K \subseteq N$

$$\begin{aligned}\sum_{j\in K} S_j(t)p_j(t) &= \sum_{j\in K}\left(\sum_{k\in N\setminus\{j\}} d_{kj}(t)p_k(t)\right)p_j(t)\\ &\geq \sum_{j\in K}\left(\sum_{k\in K\setminus\{j\}} d_{kj}(t)p_k(t)\right)p_j(t)\\ &= \sum_{j\in K}\sum_{\substack{k\in K\\k<j}} d_{kj}(t)p_k(t)p_j(t) + d_{jk}(t)p_j(t)p_k(t)\\ &= \sum_{j\in K}\sum_{\substack{k\in K\\k<j}} p_k(t)p_j(t)\\ &= \frac{1}{2}\left(\sum_{j\in K} p_j(t)\right)^2 - \frac{1}{2}\sum_{j\in K} p_j(t)^2 \ ,\end{aligned}$$

where the second equality is due to (3.15). So $S(t)$ satisfies (3.7) and for $K = N$ the inequality is tight, so $S(t)$ also satisfies (3.8). □

So, via (3.13), the scheduling polytope is an affine image of the linear ordering polytope. Lemma 3.1 is crucial for what follows, as we can continue to work with the relaxation, (3.14)-(3.15) and (3.18), instead of (3.14)-(3.17).

3.2.2 Relaxation & compactification

A linear relaxation of the optimal mechanism design problem (3.9)-(3.17) is obtained by dropping the last two sets of constraints (3.16) and (3.17) and adding (3.18). By moving from the ILP formulation to its LP relaxation, we in fact move from deterministic scheduling rules to randomized ones. Actually, the proof of Lemma 3.1 never uses that the vertices of the scheduling polytope satisfy the triangle inequality, (3.16). This shows that the triangle inequality is redundant and that (3.13)-(3.15) and (3.18) exactly describe the scheduling polytope.

In what follows we also combine (3.12) and (3.13) into just one constraint, and (3.16) and (3.17) are omitted. This gives us the following formulation for the linear relaxation.

Linear program 3

$$\min \sum_{j \in N} \sum_{i \in T_j} \varphi_j^i \pi_j^i \tag{3.19}$$

$$\pi_j^i \geq w_j^i ES_j^i \qquad \forall j \in N, i \in T_j \tag{3.20}$$

$$\pi_j^i \geq \pi_j^{i'} - w_j^i (ES_j^{i'} - ES_j^i) \qquad \forall j \in N, i, i' \in T_j \tag{3.21}$$

$$ES_j^i = \sum_{t_{-j} \in T_{-j}} \sum_{k \in N} \varphi_{-j}(t_{-j}) d_{kj}(t_j^i, t_{-j}) p_k(t_{-j}) \qquad \forall j \in N, i \in T_j \tag{3.22}$$

$$d_{jj}(t) = 0 \qquad \forall j \in N, t \in T \tag{3.23}$$

$$d_{kj}(t) + d_{jk}(t) = 1 \qquad \forall j, k \in N, k \neq j, t \in T \tag{3.24}$$

$$d_{kj}(t) \in [0,1] \qquad \forall j, k \in N, t \in T \ . \tag{3.25}$$

We now focus on the projection to variables ES_j^i, that is, vectors $ES \in \mathbb{R}^m$ satisfying (3.22)-(3.25). These are interim schedules in the linear relaxation. Let us refer to this projection as the *relaxed interim scheduling polytope*. Notice that, even though it is a linear relaxation, (3.22)-(3.25) is still an exponential size formulation in general, as it depends on the size of the type space T. The crucial insight is that, in the linear relaxation, this exponential size formulation is actually not necessary. Instead of using $d_{kj}(t)$ where $t \in T$, we propose an *LP compactification* by restricting to variables

$$d_{kj}(t_k, t_j) \ ,$$

where t_k and t_j are the types of jobs k and j, respectively. This reduces the number of d_{kj}-variables to $\mathrm{O}(m^2)$, yielding a polynomial size formulation. Doing so, we obtain

$$ES_j^i = \sum_{k \in N} \sum_{t_k \in T_k} \varphi(t_k) d_{kj}(t_j^i, t_k) p_k(t_k) \qquad \forall j \in N, t_j^i \in T_j \tag{3.26}$$

$$d_{jj}(t_j, t_j) = 0 \qquad \forall j \in N, t_j \in T_j \tag{3.27}$$

$$d_{kj}(t_k, t_j) + d_{jk}(t_j, t_k) = 1 \qquad \forall j, k \in N, k \neq j, t_j \in T_j, t_k \in T_k \tag{3.28}$$

$$d_{kj}(t_k, t_j) \in [0,1] \qquad \forall j, k \in N, t_j \in T_j, t_k \in T_k \ . \tag{3.29}$$

The following lemma is the core technical insight of the main result in this chapter.

Lemma 3.2. *The relaxed interim scheduling polytope defined by* (3.22)-(3.25) *can be equivalently described by* (3.26)-(3.29).

Proof. Let P be the projection of (3.22)-(3.25) to variables ES_j^i, and P' be the projection of (3.26)-(3.29) to variables ES_j^i. It is obvious that if $ES \in P'$, then $ES \in P$, simply by letting $d_{kj}(t) = \hat{d}_{kj}(t_k, t_j)$, for all $t \ni t_k, t_j$. So all we need to show is that, if $ES \in P$, then $ES \in P'$. Let $ES \in P$ with corresponding $d_{kj}(t)$. Now define

$$\hat{d}_{kj}(t_k, t_j) = \sum_{t \ni t_k, t_j} \frac{\varphi(t)}{\varphi_k(t_k)\varphi_j(t_j)} d_{kj}(t) \ , \tag{3.30}$$

as the weighted average of the values $d_{kj}(t)$ for those type vectors in which jobs k and j have type t_k and t_j respectively. Recall that $\varphi(t) = \varphi_1(t_1) \cdot \ldots \cdot \varphi_n(t_n)$. Since the probability distributions of all jobs are independent, we have

$$\frac{\varphi(t)}{\varphi_k(t_k)\varphi_j(t_j)} = \mathbb{P}(t|t_k, t_j) \ ,$$

the conditional probability of type vector t occurring, given that jobs k and j have type t_k and t_j. So (3.30) describes a convex combination of the $d_{kj}(t)$ variables. Thus, the $\hat{d}_{kj}(t_k, t_j)$ variables satisfy (3.27)-(3.29). Moreover, we have for all $j \in N$ and $i \in T_j$,

$$\begin{aligned} ES_j^i &= \sum_{t_{-j} \in T_{-j}} \sum_{k \in N} \varphi(t_{-j}) d_{kj}(t_j^i, t_{-j}) p_k(t_{-j}) \\ &= \sum_{k \in N} \sum_{t \ni t_j^i} \frac{\varphi(t)}{\varphi(t_j^i)} d_{kj}(t) p_k(t) \\ &= \sum_{k \in N} \sum_{t_k \in T_k} \varphi(t_k) \sum_{t \ni t_k, t_j^i} \frac{\varphi(t)}{\varphi(t_j^i)\varphi(t_k)} d_{kj}(t) p_k(t_k) \\ &= \sum_{k \in N} \sum_{t_k \in T_k} \varphi(t_k) \hat{d}_{kj}(t_k, t_j) p_k(t_k) \ , \end{aligned}$$

which is exactly the RHS of (3.26). □

We conclude with the following theorem.

Theorem 3.3. *Computing an optimal interim schedule together with optimal payments for the mechanism design problem can be done in time polynomial in the input size of the problem.*

Proof. The input size of the problem is $\Theta(m)$. The linear formulation (3.19)-(3.21) together with (3.26)-(3.29) has $O(m^2)$ variables and $O(m^2)$ constraints. Hence, this linear program can be solved in time polynomial in the input size. □

Now that we can efficiently compute an interim scheduled and corresponding optimal payments, two issues remain: The first is the interpretation of Theorem 3.3, because it is based on a relaxation and has a reduced number of variables. The second is the actual implementation of the optimal mechanism: We have to link the computed solution of the LP relaxation, specifically the computed interim schedule ES, to a (randomized) schedule $S(t)$ for any given type profile $t \in T$. The first issue is discussed next, the second is treated separately in Section 3.3.

3.2.3 Discussion of the result in Theorem 3.3

We consider a true relaxation of the linear ordering polytope by dropping triangle and integrality constraints, yet the affine image of the variables $d_{kj}(t)$, respectively

$\hat{d}_{kj}(t_k, t_j)$, via (3.13) still yields a feasible point in the scheduling polytope. This allows us to interpret the solution as a (randomized) schedule; this is discussed in the next section. Also, we have drastically reduced the number of variables. It seems that thereby we are reducing the (number of) feasible mechanisms, because the variables $\hat{d}_{kj}(t_k, t_j)$ only depend on the types of jobs k and j, while $d_{kj}(t)$ depends on the whole type vector t. For deterministic mechanisms, this is also known as the *IIA-property* [33, 19].

Definition 3.1 (iia)**.** A deterministic scheduling rule is *independent of irrelevant alternatives*, or IIA, if the relative order of two jobs does not depend on anything but the types of those two jobs, that is, $d_{kj}(t) = \hat{d}_{kj}(t_k, t_j)$, for all $t \ni t_k, t_j$. We call a mechanism for which the scheduling rule is IIA, an IIA-*mechanism.*

Lemma 3.2 shows that the reduction of variables is in fact no loss of generality for the linear relaxation. Interestingly, it *is* a loss of generality for the linear ordering polytope itself, respectively for the deterministic optimal mechanism design problem (3.9)-(3.17): Duives et al. [19] give an instance that shows the existence of an optimality gap in general. That is, there exist instances where the optimal IIA mechanism has higher total expected cost than the optimal mechanism; see also Theorem 3.4 below. With this in mind, a possible interpretation of Lemma 3.2 would be that the restriction to IIA-mechanisms is no loss of generality once randomization is allowed. But this interpretation is problematic too, as the variables d_{kj} in the relaxation cannot, in general, be interpreted as the probability of Job k preceding Job j: By definition of the relaxation, neither the vector of variables $\hat{d}_{kj}(t_k, t_j)$ nor $d_{kj}(t)$ do necessarily lie in the linear ordering polytope; see e.g. Fishburn [23]. In the next section we discuss how to deal with this problem.

3.3 Implementation of the optimal mechanism

Recall from the previous discussion that the fractional solution in variables d_{kj} as suggested by the LP relaxation cannot in general be decomposed into linear orders, as it may lie outside the linear ordering polytope. Still, by taking the detour via the scheduling polytope, we can fix this problem.

Observe that, for a given solution of the LP relaxation and any fixed type vector $t = (t_1, \ldots, t_n)$, we have values $\hat{d}_{jk}(t_j, t_k)$, for each pair of jobs j and k. From these we can compute a corresponding vector of start times $S(t)$ by

$$S_j(t) = \sum_{k \in N} d_{kj}(t_j, t_k) p_k(t_k) \text{ for all } j \ .$$

Now $S(t)$ is a point in the scheduling polytope defined in (3.7) and (3.8). The dimension of the scheduling polytope for n jobs is $n - 1$. Caratheodory's Theorem says that any point in a d-dimensional polytope can be expressed as a convex combination of at most $d + 1$ of the vertices of that polytope [9]. Therefore $S(t)$ can be expressed as the convex combination of at most n vertices of the scheduling polytope, that is, deterministic schedules. The results from Chapter 5 show that there are fast algorithms to find such a convex combination of vertices for any point in the scheduling polytope. Combining these with the outcomes from the LP results in a method that, for any type vector $t \in T$, finds a lottery over pure schedules and a payment vector, which combined are a Bayes-Nash incentive compatible mechanism for the two-dimensional scheduling mechanism design problem.

3.4 Computational results

We have implemented all integer linear programming models discussed in this chapter. Here we describe the results obtained with these experiments.

We compute both optimal mechanisms and optimal mechanisms that are in addition IIA. To this end, for each pair of type vectors t, t', with $t_i = t'_i$ and $t_j = t'_j$, and each pair of permutations σ, σ' that do not agree on the order of i and j, we add the following constraint to (3.1)-(3.6):

$$x(\sigma, t) + x(\sigma', t') \leq 1 \ . \tag{3.31}$$

This constraint ensures that if the types of i and j stays the same, then their relative order stays the same. To ensure the IIA condition in the extended linear ordering formulation, we simply reduce the number of variables by letting $\delta_{ij}(t)$ only depend on the types of i and j, i.e., by using variables $\delta_{ij}(t_i, t_j)$ instead.

As mentioned, the most straightforward ILP formulation (3.1)-(3.6) for the deterministic mechanism design problem is colossal, which is confirmed by large computation times. In comparison, the linear ordering formulation (3.9)-(3.17), even though exponential in size as well, yields a substantial improvement in computation times even for small scale instances. In particular, the latter allows a drastic reduction of the number of variables and constraints for computing IIA-mechanisms, while in the former the number of variables remains the same and the number of constraints actually increases drastically. Tables 3.1 to 3.4 show the computational results for the different formulations for some small scale instances[3]. We see that the formulation in natural variables (3.1)-(3.6) is clearly outperformed by the extended formulation in linear ordering variables, (3.9)-(3.17). Especially when the number of jobs and/or types per job increases. The difference in performance is obvious. In particular, for computing mechanisms that are IIA, the advantage of the extended formulation in linear ordering variables is overwhelming.

The main purpose of implementing the integer linear programming models was, however, not to verify what was to be expected. It was to verify certain conjectures about the relations between different classes of mechanisms through testing randomly generated instances. The following two of these instances lead to some new insights.

Instance 3.1. Four jobs with the following type spaces and corresponding probabilities:

[3]Computation times where obtained with Gurobi 5.6.3 64-bit in Python 3.2.5 running on an Intel(R) Core(TM)2 Duo E8400 3.00 GHz computer with Windows 7 64-bit operating system with 4.00 GB of memory.

Types	Setting	Formulation	# Vars	# Constrs	Comp. time (ms)
3-3-3	BN	(3.1)-(3.6)	180	69	5.70
3-3-3	BN-IIA	(3.1)-(3.6),(3.31)	180	1527	22.20
3-3-3	BN	(3.9)-(3.17)	99	96	3.13
3-3-3	BN-IIA	(3.9)-(3.17)	45	96	1.87

Table 3.1: Computational results for 30 randomly constructed instances with three jobs, each with three types.

Types	Setting	Formulation	# Vars	# Constrs	Comp. time (ms)
3-3-3-3	BN	(3.1)-(3.6)	1968	138	66.67
3-3-3-3	BN-IIA	(3.1)-(3.6),(3.31)	1968	560010	75900.03
3-3-3-3	BN	(3.9)-(3.17)	510	705	15.50
3-3-3-3	BN-IIA	(3.9)-(3.17)	78	273	5.10

Table 3.2: Computational results for 30 randomly constructed instances with four jobs, each with three types.

Job 1	$w = 6$	$w = 7$	$w = 10$
$p = 2$	0.3312	0.3456	0.0432
$p = 7$	0.1288	0.1344	0.0168

,

Job 2	$w = 5$	$w = 8$
$p = 4$	0.0344	0.8256
$p = 8$	0.0056	0.1344

,

Job 3	$w = 3$	$w = 10$
$p = 8$	0.3825	0.1275
$p = 10$	0.3675	0.1225

,

Job 4	$w = 3$	$w = 8$
$p = 1$	0.2583	0.3717
$p = 6$	0.1517	0.2183

.

Instance 3.2. Three jobs with the following type spaces and corresponding probabilities:

Job 1	$w = 2$
$p = 1$	1

,

Job 2	$w = 9$
$p = 8$	1

,

Job 3	$w = 1$	$w = 3$	$w = 5$
$p = 5$	0.24	0.02	0.16
$p = 7$	0.24	0.24	0.10

.

Note that Instance 3.2 is an instance that was found by Duives et al. [19]. They use it to prove that optimal Bayes-Nash mechanisms do not satisfy the IIA condition. As a matter of fact, Instance 3.1 shows that also in the dominant strategy setting, optimal mechanisms do not satisfy the IIA condition. Another commonly asked question is if optimal Bayes-Nash mechanisms can be implemented in dominant strategies. For scheduling in the single-dimensional setting, this is indeed the case [19]. Our computations on Instance 3.1 show that, for the two-dimensional setting, the same does not hold. Finally, Instance 3.2 shows that the resulting randomized mechanism from the LP formulation in Section 3.2 can not be implemented deterministically. We therefore obtain the following theorems.

Types	Setting	Formulation	# Vars	# Constrs	Comp. time (ms)
6-6-6	BN	(3.1)-(3.6)	1332	327	43.57
6-6-6	BN-IIA	(3.1)-(3.6),(3.31)	1332	29487	710.37
6-6-6	BN	(3.9)-(3.17)	684	543	22.97
6-6-6	BN-IIA	(3.9)-(3.17)	144	543	9.57

Table 3.3: Computational results for 30 randomly constructed instances with three jobs, each with six types.

Types	Setting	Formulation	# Vars	# Constrs	Comp. time (ms)
9-9-9	BN	(3.1)-(3.6)	4428	945	404.65
9-9-9	BN-IIA	(3.1)-(3.6),(3.31)	4428	158409	22506.73
9-9-9	BN	(3.9)-(3.17)	2241	1674	257.91
9-9-9	BN-IIA	(3.9)-(3.17)	297	1674	152.41

Table 3.4: Computational results for 30 randomly constructed instances with three jobs, each with nine types.

Theorem 3.4. *Optimal deterministic mechanisms for both Bayes-Nash and dominant strategy implementations, in general do not satisfy the IIA condition.*

Proof. Duives et al. [19] use Instance 3.2 to prove this theorem for Bayes-Nash mechanisms[4]. Instance 3.1 shows the same: it has a Bayes-Nash optimal IIA mechanism with objective value 128.5697 and non-IIA optimal Bayes-Nash mechanism with objective value 128.5195. Moreover, computations for Instance 3.1 also show that dominant strategy optimal mechanisms yield an objective value 128.6946 for the IIA mechanism, while for the non-IIA mechanism, we obtain an objective value of 128.6151. □

Theorem 3.5. *The optimal deterministic Bayes-Nash mechanism is generally not implementable in dominant strategies.*

Proof. Instance 3.1 has optimal deterministic Bayes-Nash mechanism with objective value 128.5195, while the optimal deterministic dominant strategy mechanism has objective value 128.6151. □

Theorem 3.6. *Randomized Bayes-Nash mechanisms perform better than deterministic Bayes-Nash mechanisms in terms of total optimal payment.*

Proof. Instance 3.2 has a deterministic Bayes-Nash optimal mechanism with objective value 45.0, while the randomized Bayes-Nash optimal mechanism has objective value 44.74625. □

[4]Note that the example given in Heydenreich et al. [33] to prove the same theorem was flawed, but that problem has been fixed in [19].

3.4.1 The IIA condition and the constraint matrix

When we apply our compactification to the deterministic problem, i.e. the integer program, this results in an integer program that finds the optimal IIA deterministic mechanism. While for the relaxed problem this is without loss of generality, the deterministic problem in general does not have an optimal solution that satisfies the IIA property. One of the possible explanations for this is the structure of the constraint matrix for both problems. Although, we refer to the non-deterministic problem as the LP-relaxation of the IP, there is more going on than only relaxing the integer variables. As mentioned in Section 3.2.2, the triangle inequality, (3.16), is redundant once we look at the LP-relaxation. Therefore, it suffices to use inequalities (3.22)-(3.25). These constraints result in a constraint matrix with a very specific structure, namely the matrix is block diagonal. Furthermore, variables $d_{jk}(t)$ only play a roll in the expected start time of agents j and k, but no other agent. Thus, these decision variables need only depend on the types of those two agents and the compactification can be done without affecting the outcome. While this all works for the linear relaxation, adding the triangle inequality which is necessary for deterministic mechanisms, breaks this structure. Hence, for deterministic mechanisms, the decision variables need to be dependent on the whole type vector t.

CHAPTER 4

Heuristics for deterministic mechanism design

Chapter 3 shows that the two-dimensional scheduling mechanism design problem can be solved in polynomial time for randomized mechanisms by using an LP formulation. Still, it is unsatisfying that this does not get us much closer toward deterministic mechanisms. In this chapter we provide computationally efficient heuristics that find mechanisms that are both deterministic and simple to interpret. The latter is achieved by enforcing the, so-called, IIA property on the mechanism. This ensures that the allocation rule of the mechanism can be viewed as a priority list of the types of all the jobs. On the one hand, this makes sense because the relative order of two jobs in the schedule only depends on their respective types. On the other hand, we show that the IIA property sets the problem in the class $\mathcal{NP}$, which is unknown to be true for the general problem.

While for many problems it is feasible to look at rounding procedures to transform randomized solutions into deterministic ones, this does not seem feasible for the two-dimensional scheduling mechanism design problem. For example, if we would round the linear ordering variables, from LP formulation 3, to the nearest binary value, in general, this would not lead to deterministic schedules, since the outcome may not satisfy the triangle inequality.

In this chapter we consider the same model as in Chapter 3. A set $N = \{1, \dots, n\}$ of jobs have to be scheduled on a single machine. Each Job j has a private type t_j from a set of possible types T_j. Each type consists of a weight w_j and a processing time p_j. Over the set of types T_j a probability distribution, $\varphi_j : T_j \to [0,1]$ is publicly known. A mechanism consists of a pair (f, π), where f is the scheduling rule and π is a vector of payments, $\pi_j(t_j)$ for each Job j and each type $t_j \in T_j$. Our objective is to find a deterministic mechanism that is BNIC and IR, while minimizing the expected total payments made to the jobs.

In general, deterministic mechanisms may need an exponential size description, since for every type vector a schedule must be determined. Hence, it is not clear if the problem is contained in $\mathcal{NP}$. Also, this makes local search heuristics very hard to apply, since changes in the schedule for one type vector influence the outcome very little. In this chapter we consider IIA mechanisms to tackle these problems. We show that IIA mechanisms have a polynomial size representation, that additionally makes the mechanism easy to interpret. It follows that the deterministic IIA scheduling mechanism design problem is contained in $\mathcal{NP}$.

In the single-dimensional problem there is always an optimal solution that satisfies the IIA condition [19]. However, for the two-dimensional problem, we know that the optimal solution, in general, does not satisfy this condition.

We propose a representation of IIA mechanisms as a priority list of all types. This priority list is easy to interpret and makes local search methods viable. With respect to this priority list, we propose two different search neighborhoods to use in the local search heuristics. We show with computational experiments, that one of the neighborhoods allows the local search heuristics to find good solutions. However, this requires a relatively long computation time. Hence, the computational results show that the local search heuristics are not well-suited for larger problem instances. Therefore, we developed several experimental heuristics based on the solution concept for the single-dimensional mechanism design problem. The experiments show that these heuristics find good deterministic solutions, in much less time even than it takes commercial software to solve the LP relaxation from Chapter 3.

As far a we are aware, there has not been much work toward heuristics for optimal mechanism design. In this chapter we suggest some techniques that seem promising for problems of which the allocation can be constructed from a simple ordering of the agents. These include scheduling problems, and single item auctions and many multi item auctions could be modeled in such a way as well.

4.1 Preliminary results

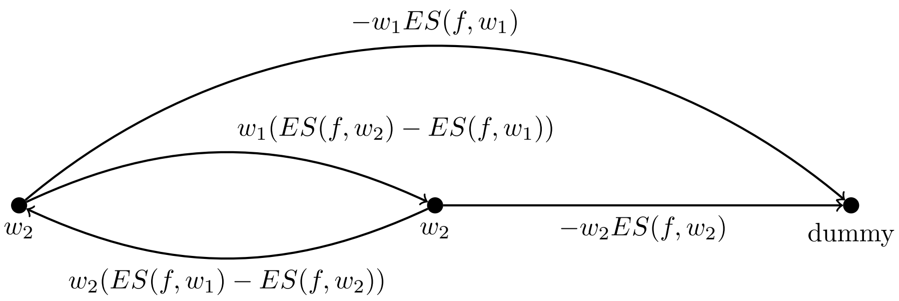
Duives et al. [19] treat mechanism design in a single machine scheduling setting where the jobs have private weights but processing times are public knowledge. They show that some of the results are transferable to the problem we consider, while some are not. In this section we recap some of the results from Duives et al. [19] that we will use in the rest of the chapter.

The main result from [19] is that, like the auction setting treated by Myerson [49], the single-dimensional scheduling mechanism design problem can be solved by means of solving the standard optimization problem on so-called *virtual weights*. These virtual weights can be computed from the *type graph* of each of the jobs.

The type graph of Player j can be used as a tool to compute minimal payments given an implementable allocation rule and can be constructed as follows. For Player j, the type graph is a complete directed graph, whose nodes consist of all $t_j \in T_j$. Given an allocation rule f, from which we compute for each type t_j the expected start time $ES_j(f, t_j)$, and two types $t_j \neq t'_j$, let

$$l(f, t_j, t'_j) = w_j(t_j)(ES_j(f, t'_j) - ES_j(f, t_j))$$

be the length of the arc (t_j, t'_j). These lengths represent the gain in expected valuation for Player j if he truthfully reports type t_j instead of lying type t'_j. The type graph has one more *dummy node*, which has incoming arcs from all other nodes and



$-w_1 ES(f, w_1)$
$w_1(ES(f, w_2) - ES(f, w_1))$
w_2
w_2
$-w_2 ES(f, w_2)$
dummy
$w_2(ES(f, w_1) - ES(f, w_2))$

Figure 4.1: Type graph for a job with two types, (w_1, p) and (w_2, p).

no outgoing arcs. The length of the incoming arcs is

$$l(f, t_j, \text{‘dummy’}) = -w_j(t_j) ES_j(f, t_j) \ .$$

Figure 4.1 shows the type graph for a job with two types. Given these arc lengths, the BNIC constraints, (1.14), can be rewritten as

$$\begin{aligned} E\pi_j(f, t'_j) &\geq E\pi_j(f, t_j) + w_j(t_j) ES_j(f, t'_j) - w_j(t_j) ES_j(f, t_j) \\ &= E\pi_j(f, t_j) + l(f, t_j, t'_j) \ , \end{aligned}$$

from which we see that $E\pi_j(f, \cdot)$ is a node potential in the type graph of Job j. This implies that the allocation rule f is Bayes-Nash implementable if and only if none of the resulting type graphs has a negative cycle. Such an allocation rule is said to be *cyclically monotone* [10]. For mechanism design problems where the agents have quasi-linear utilities, am allocation rule is cyclically monotone if and only if it is *2-cycle monotone*, that is, the type graphs have no negative two-cycles [62].

We say an allocation rule, f, for the single-dimensional scheduling mechanism design problem, satisfies *monotonicity* if for every Job j and every pair of types $t_j, t'_j \in T_j$, such that $w_j(t_j) > w_j(t'_j)$, we have $ES_j(f, t_j) \leq ES_j(f, t'_j)$.

Theorem 4.1 (Duives et al. [19]). *An allocation rule f for the single-dimensional scheduling mechanism design problem is Bayes-Nash implementable if and only if it satisfies monotonicity.*

Proof. An allocation rule f for the scheduling mechanism design problem is 2-cycle monotone if the following is non-negative for all t_j and t'_j,

$$\begin{aligned} l(f, t_j, t'_j) + l(f, t'_j, t_j) &= w_j(t_j)(ES_j(f, t'_j) - ES_j(f, t_j)) \\ &\quad + w_j(t'_j)(ES_j(f, t_j) - ES_j(f, t'_j)) \\ &= (w_j(t_j) - w_j(t'_j))(ES_j(f, t'_j) - ES_j(f, t_j)) \ . \end{aligned} \tag{4.1}$$

Obviously, this is true if and only if f satisfies monotonicity. □

By normalizing the node potential of the dummy node to 0, the resulting minimal node potentials, $E\pi_j(f,\cdot)$, are exactly the minimum payments for any given allocation rule, f. The minimum payments for a type t_j can then be computed by finding the shortest path from t_j to the dummy node in the type graph of j. Let $\text{dist}(f, t_j, \text{dummy})$ denote the length of this shortest path. Then, for allocation rule f, the minimum expected payment, $E\pi_j(f,t_j)$, for t_j, such that $(f, E\pi)$ is BNIC and IR is

$$E\pi_j(f,t_j) = -\text{dist}(f, t_j, \text{dummy}) \ .$$

This is a well-known result and also holds for the two-dimensional case we treat in this chapter [49, 62]. For the single-dimensional case let $T_j = \{t_j^1, \ldots, t_j^{\tau_j}\}$ and $|T_j| = \tau_j$, such that $w_j^1 < w_j^2 < \ldots < w_j^{\tau_j}$, where $w_j^i = w_j(t_j^i)$ as shorthand notation. Then for any monotone allocation rule f and for all $j \in N$ and $t_j^i, t_j^{i'} \in T_j$, $i+1 < i'$

$$\begin{aligned}
l(f, t_j^i, t_j^{i'}) &= w_j^i(ES_j(f, t_j^{i'}) - ES_j(f, t_j^i)) \\
&= w_j^i(ES_j(f, t_j^{i'}) - ES_j(f, t_j^{i+1})) + w_j^i(ES_j(f, t_j^{i+1}) - ES_j(f, t_j^i)) \\
&\leq w_j^i(ES_j(f, t_j^{i'}) - ES_j(f, t_j^{i+1})) + w_j^{i+1}(ES_j(f, t_j^{i+1}) - ES_j(f, t_j^i)) \\
&= l(f, t_j^i, t_j^{i+1}) + l(f, t_j^{i+1}, t_j^{i'}) \ ,
\end{aligned}$$

since $ES_j(f, t_j^{i+1}) - ES_j(f, t_j^i) \leq 0$ from monotonicity of f. Thus, for any Bayes-Nash implementable allocation rule, f, and for all $j \in N$ and $t + j^i \in T_j$,

$$\text{dist}(f, t_j^i, \text{dummy}) = l(f, t_j^i, t_j^{i+1}) + \text{dist}_f(t_j^{i+1}, \text{dummy}) \ . \tag{4.2}$$

This shows that, for any Bayes-Nash implementable allocation rule f, when computing the shortest paths, it suffices to consider the *reduced type graph*, where only arcs between two subsequent types exist. Recursing on (4.2) yields

$$\begin{aligned}
\text{dist}(f, t_j^i, \text{dummy}) &= l(f, t_j^i, t_j^{i+1}) + \ldots + l(f, t_j^{\tau_j}, \text{dummy}) \\
&= \sum_{k=i}^{\tau_j - 1} \left(w_j^k(ES_j(f, t_j^{k+1}) - ES_j(f, t_j^k)) \right) - w_j^{\tau_j} ES_j(f, t_j^{\tau_j}) \\
&= -w_j^i ES_j(f, t_j^i) + \sum_{k=i+1}^{\tau_j} (w_j^{k-1} - w_j^k) ES_j(f, t_j^k) \ .
\end{aligned}$$

From this we see that we can write the objective function, minimizing total payments, as

$$\begin{aligned}
\sum_{j \in N} \sum_{t_j \in T_j} E\pi_j(f, t_j) &= \sum_{j \in N} \sum_{i=1}^{\tau_j} \varphi_j(t_j^i) w_j^i + \sum_{k=1}^{i-1} \varphi_j(t_j^k)(w_j^{i-1} - w_j^i) ES_j(f, t_j^i) \\
&= \sum_{j \in N} \sum_{i=1}^{\tau_j} \varphi_j(t_j^i) \overline{w}_j^i ES_j(f, t_j^i) \ ,
\end{aligned} \tag{4.3}$$

where $\overline{w}_j^i$ are called the *virtual weights*, with $\overline{w}_j^1 = w_j^1$ and

$$\overline{w}_j^i = w_j^i + \frac{\sum_{k=1}^{i-1} \varphi_j(t_j^k)}{\varphi_j^i}(w_j^{i-1} - w_j^i) \qquad \text{for all } i = 2, \dots, \tau_j \ . \tag{4.4}$$

It follows that Smith's rule, with respect to these virtual weights, scheduling the jobs in non-increasing order of $\overline{w}_j(t_j)/p_j$, minimizes the total expected payments, (4.3). Therefore, as long as the resulting allocation rule satisfies monotonicity, Smith's rule with respect to the virtual weights is also the optimal allocation rule. Let F_{mon} denote the space of monotone allocations. If Smith's rule, with respect to the virtual weights, does not satisfy monotonicity, it follows that for the optimization problem

$$\min_{f \in F_{\text{mon}}} \sum_{j \in N} \sum_{i=1}^{\tau_j} \varphi_j(t_j^i) \overline{w}_j^i ES_j(f, t_j^i) \ ,$$

some of the monotonicity constraints, $ES_j(f, t_j^1) \leq \ldots \leq ES_j(f, t_j^{\tau_j})$, are binding. These must be those for which $ES_j(f, t_j^i) < ES_j(f, t_j^{i+1})$, when scheduled according Smith's rule, with respect to the virtual weights. In this case, the optimal allocation rule can still be described in the same way, scheduling the jobs in non-increasing order of $\overline{w}_j(t_j)/p_j$, after a procedure known as ironing. This is explained next.

Definition 4.1 (Ironing)**.** Let a single-dimensional scheduling mechanism design problem and virtual weights as defined by (4.4) be given. Let $w_j^i < w_j^{i+1}$ for all jobs j and types i. For Job j let i be the largest index such that

$$\overline{w}_j^i < \overline{w}_j^{i-1} \ . \tag{4.5}$$

For this pair $i, i-1$ let the ironed virtual weight be

$$\overline{w}_j^{i,i-1} = \frac{\varphi_j(w_j^i)\overline{w}_j^i + \varphi_j(w_j^{i-1})\overline{w}_j^{i-1}}{\varphi_j(w_j^i) + \varphi_j(w_j^{i-1})} \ ,$$

for both i and $i-1$. Repeat this process until no index satisfies (4.5).

Theorem 4.2. *For the single-dimensional scheduling mechanism design problem, scheduling the jobs in order of non-increasing ratio of ironed virtual weights over processing time is the optimal allocation rule.*

Proof. We know that scheduling the jobs in non-increasing ratio of virtual weights over processing time minimizes (4.3). However, if the outcome does not satisfy monotonicity, then for some i we have, $ES_j^i > ES_j^{i+1}$, which violates monotonicity. Therefore, we minimize (4.3) with $ES_j^i \leq ES_j^{i+1}$ as an extra constraint, which we know will be a binding constraint. Therefore, the optimal solution must satisfy $ES_j^i = ES_j^{i+1}$. Now, since types i and $i+1$ are treated equally by the allocation rule, we can treat them as a single type, i', for which the job can always report the higher weight, w_j^{i+1}. A simple computation then yields that the virtual weight of such a type is

$$\begin{aligned}
\overline{w}_j^{i'} &= w_j^{i+1} + (w_j^{i+1} - w_j^{i-1})\frac{\sum_{k=1}^{i-1}\varphi_j(t_j^k)}{\varphi_j^i + \varphi_j^{i+1}} \\
&= w_j^{i+1} + \frac{(w_j^i - w_j^{i+1})\varphi_j^i}{\varphi_j^i + \varphi_j^{i+1}} \\
&\quad + \frac{(w_j^{i+1} - w_j^i)(\varphi_j^i + \sum_{k=1}^{i-1}\varphi_j(t_j^k)) + (w_j^i - w_j^{i-1})\sum_{k=1}^{i-1}\varphi_j(t_j^k)}{\varphi_j^i + \varphi_j^{i+1}} \\
&= \frac{\varphi_j^{i+1} w_j^{i+1}}{\varphi_j^i + \varphi_j^{i+1}} + \frac{\varphi_j^i w_j^i}{\varphi_j^i + \varphi_j^{i+1}} \\
&\quad + \frac{(w_j^{i+1} - w_j^i)(\sum_{k=1}^{i}\varphi_j(t_j^k)) + (w_j^i - w_j^{i-1})\sum_{k=1}^{i-1}\varphi_j(t_j^k)}{\varphi_j^i + \varphi_j^{i+1}} \\
&= \frac{\varphi_j^{i+1}\overline{w}_j^{i+1} + \varphi_j^i\overline{w}_j^i}{\varphi_j^i + \varphi_j^{i+1}} \quad ,
\end{aligned}$$

which is exactly the ironed weight for types i and $i+1$. We can repeat this until no subset satisfies (4.5). Note that, if this is the case, the allocation rule, that schedules all jobs in non-decreasing order of virtual weight over processing time, must satisfy monotonicity. Finally, if (4.5) holds for some subset of types, while the corresponding allocation rule does satisfy monotonicity then the expected start times for the whole subset of types must be equal. In that case, applying ironing can not change the allocation rule, as both the highest and the lowest virtual weight in that subset result in the same expected start time. Thus, the average must also result in that expected start time. □

For the two-dimensional scheduling mechanism design problem we can still use the same graph interpretation of BNIC. The type graph can be constructed in the same way and we can compute the minimum payments in the same way from the shortest paths lengths in this type graph. Also, Duives et al. [19] show that the type graph can be reduced, like in Figure 4.2. In this graph, though, there are still several possible shortest paths to the dummy node, and not one of them is dominant in the sense that it is always a shortest path. Because of this, directly applying the virtual weights method from the single-dimensional case is doomed to fail.

In the remainder of this chapter we use generalized versions of the "virtual weights method" from the single-dimensional case in two ways. The first is to apply the method on a processing time by processing time basis, ignoring all types with other processing times. We refer to this as *1D virtual weights*. The second method, which we refer to as *computing virtual weights with respect to some shortest paths*, is the

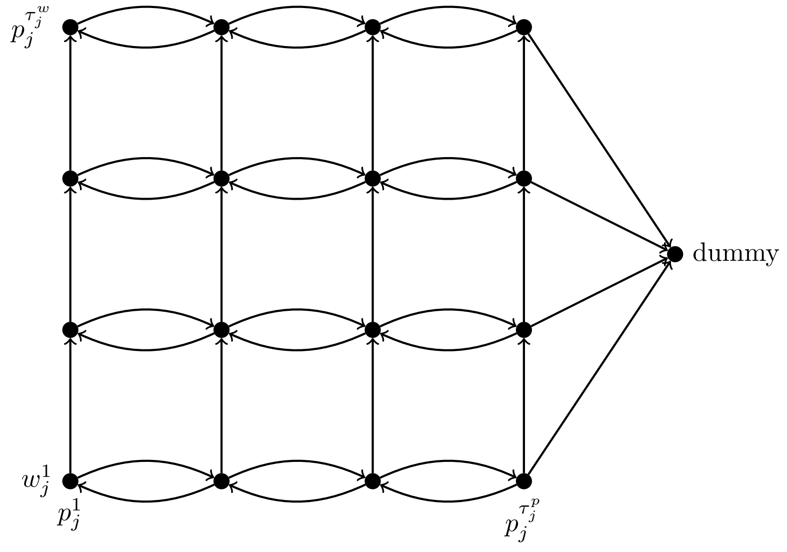


Figure 4.2: Generic reduced type graph for two-dimensional types.

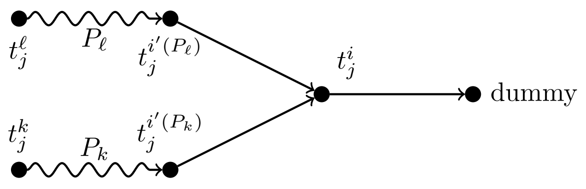


Figure 4.3: For each shortest path, P_k (P_ℓ), through t_j^i we add $\frac{\varphi_j^k}{\varphi_j^i}(w_j^{i'(P_k)} - w_j^i)$ to the virtual weight of i.

following. Given shortest paths, P_j, compute virtual weights for all types as follows

$$\overline{w}_j^i = w_j^i + \frac{\sum_{k:i \in P_k} \varphi_j^k}{\varphi_j^i}(w_j^{i'(P_k)} - w_j^i) \ ,$$

here $i \in P_k$ means that type i is in the shortest path of type k and $i'(P_k)$ denotes the predecessor of i in that path. In Figure 4.3 this idea is visualized. Note that this is indeed a generalization of the virtual weights for the single-dimensional problem.

The generalized virtual weight methods for the two-dimensional problem are used in Section 4.3 and Section 4.4. The "1D virtual weights"-method is used as a base solution for local search heuristics and the "virtual weights with respect to some shortest paths"-method is used for a heuristic that iteratively improves an incumbent solution.

4.2 IIA mechanisms

For the two-dimensional scheduling mechanism design problem, for deterministic non-IIA mechanisms it is not clear if, given a scheduling rule, the expected start time for Job j and type t_j, can be computed in polynomial time. Therefore, we do not know if the decision problem, asking if there exists a mechanism with expected total payment no more than Π, for some $\Pi \in \mathbb{R}_+$, is contained in $\mathcal{NP}$. Consider the expected start time, ES_j^i, of Job j in type t_j^i. The fact that the mechanism is not IIA, exactly expresses that the relative order of any two jobs may depend on the types of other jobs. Hence, computing the expected start time for a Job j in one type i, ES_j^i, may require the consideration of a number of type vectors that is exponential in the number of jobs.

Unlike general deterministic mechanisms, IIA mechanisms can be succinctly represented by a priority list of all the types of all jobs. This priority list induces a linear ordering of the jobs based on the type they report. A job can easily check for each of it's own types the expected start time and for any type of the other jobs if it would by scheduled before or after.

Example 4.1. Consider an allocation rule for two jobs represented by the priority list, $(t_1^1, t_1^2, t_2^1, t_2^2, t_1^3)$, where t_j^i denotes the i-th type of Job j. In this case Job 1 is scheduled before Job 2 except when it reports type t_1^3. The expected start time of Job 2 is, independent from its reported type, $\varphi_1(t_1^1)p(t_1^1) + \varphi_1(t_1^2)p(t_1^2)$.

This representation is actually identical to the IIA formulation in linear ordering variables, as mentioned in Section 3.4.

Given any IIA mechanism, let its linear ordering be given by linear ordering variables, $d_{kj}(t_k, t_j)$, such that

$$d_{kj}(t_k, t_j) = \begin{cases} 1 & \text{if Job } k\text{, with } t_k\text{, is scheduled before Job } j\text{, with } t_j \\ 0 & \text{otherwise} \end{cases} .$$

These linear ordering variables can easily be computed from a priority list, by simply checking which type, t_k or t_j, occurs first in that list. The linear ordering variables, computed like this, satisfy

$$\begin{aligned} d_{jj}(t_j, t_j) &= 0 && \forall j \in N \\ d_{kj}(t_k, t_j) + d_{jk}(t_j, t_k) &= 1 && \forall j, k \in N, k \neq j \\ d_{kj}(t_k, t_j) + d_{j\ell}(t_j, t_\ell) &\leq 1 + d_{k\ell}(t_k, t_\ell) && \forall j, k, \ell \in N \ . \end{aligned}$$

Now, it is easy to compute the expected start time for any type of any job,

$$ES_j^i = \sum_{k \in N} \sum_{t_k \in T_k} \varphi_k(t_k) d_{kj}(t_k, t_j^i) p_k(t_k) \ .$$

To ensure that a linear ordering corresponds to a BNIC allocation rule, the resulting rule has to satisfy monotonicity. In the priority list this translates to the simple condition that for each job all types with the same processing time have to appear in descending order of their weights. It is easy to check that this condition is sufficient. It is not necessary, since two types of the same job, that appear directly after each other, have the same expected start time, independent of their relative order. Yet, it is without loss of generality, for the same reason. Let us call an ordering that satisfies this condition a *proper* ordering.

4.3 Local search heuristics for priority list mechanisms

Local search heuristics are algorithms that work on the simple idea of iteratively searching for a better solution. Such a heuristic starts with a *base solution* and, from there, iteratively searches a certain *neighborhood* of the incumbent solution, according to some selected definition of neighborhood. For example, for linear orderings, a pairwise exchange is a standard neighborhood. This underlies, for instance, the bubble sort algorithm.

Let $b(i)$ denote the incumbent solution in iteration i and $Z(b(i))$ the set of solutions in its neighborhood. A local search algorithm chooses in iteration i one solution $b(i+1) \in Z(b(i))$, the incumbent solution for the next iteration.

Solutions to the problem we consider here are represented by a linear ordering of the job types. This allows some relatively simple neighborhoods. We define the following two:

The *pairwise adjacent exchange* neighborhood is very simple. For a given solution b its neighbors are found by switching two adjacent job types in the ordering, if they are not from the same job with the same processing time.

The *pairwise exchange* neighborhood is a little bit more involved. The idea is to switch any two job types. In this case, however, any job types, that are of the same job and have the processing time as any one of those job types, need to be switched as well. This works as follows. Take any two job types t_j and t_k in the linear ordering that are not from the same job with the same processing time. Take all job types that are ordered in between t_j and t_k, that are from the same job and have the same processing time as either t_j or t_k. Now, switch t_j and t_k and order the other types directly before or after t_j or t_k such that the result is a proper ordering.

We tested both the pairwise adjacent exchange neighborhood and the pairwise exchange neighborhood for local search. We considered *steepest descent* and *random* as methods for choosing from the neighborhoods.

Steepest descent computes for all solutions $b' \in Z(b)$ the objective value and chooses that solution which has the smallest objective value, if it is smaller than that of the incumbent solution, b. With *random* we mean randomly selecting a solution from the neighborhood, until we find one that has a smaller objective value than the incumbent solution or the maximum number of tries is reached. We set the maximum tries for the random method to $10 \sum_{j\in N} |T_j| + 100$, where $\sum_{j\in N} |T_j|$ is the total number of types. This value kept the running time low, while obtaining good results in terms of objective function.

The results of our test of these local search methods are given in Tables 4.1 and 4.2. Both tables show the average results of ten instances, with ten jobs and four types and five jobs and nine types, respectively.

Starting solution	Method	Neighborhood	Obj. value/opt	Comp. time
Smith's rule	Random	Pair exchange	1,000	8,698
Smith's rule	Random	Adjacent pair	1,002	4,425
Smith's rule	Steepest	Pair exchange	1,000	68,043
Smith's rule	Steepest	Adjacent pair	1,155	0,217
1D virt. weights	Random	Pair exchange	1,001	7,785
1D virt. weights	Random	Adjacent pair	1,001	3,381
1D virt. weights	Steepest	Pair exchange	1,000	55,285
1D virt. weights	Steepest	Adjacent pair	1,039	0,169

Table 4.1: Average values over ten instances, with ten jobs, each with four types. The optimal objective value was computed with the IIA linear ordering ILP formulation, (3.9)-(3.17). The average computation time to solve the ILP was 1,034 s. Smith's rules objective value was 1,160 times the optimal value. 1D virtual weights objective value was 1,039 times the optimal value.

Starting solution	Method	Neighborhood	Obj. value/opt	Comp. time
Smith's rule	Random	Pair exchange	1,004	22,141
Smith's rule	Random	Adjacent pair	1,048	10,641
Smith's rule	Steepest	Pair exchange	1,002	160,803
Smith's rule	Steepest	Adjacent pair	1,254	0,426
1D virt. weights	Random	Pair exchange	1,002	21,584
1D virt. weights	Random	Adjacent pair	1,022	10,039
1D virt. weights	Steepest	Pair exchange	1,001	128,123
1D virt. weights	Steepest	Adjacent pair	1,122	0,488

Table 4.2: Average values over ten instances, with five jobs, each with nine types. The optimal objective value was computed with the randomized LP formulation. The average computation time to solve the LP was 0,007 s. Smith's rules objective value was on average 1,260 times the optimal value. 1D virtual weights objective value was on average 1,129 times the optimal value.

What we clearly see from these results, is that the pairwise exchange neighborhood outperforms the adjacent pairwise exchange neighborhood in terms of objective value. The adjacent pairwise exchange neighborhood seems to outperform the pairwise exchange neighborhood in terms of computation time. However, this is mostly because it hardly finds any improvements on the starting solution. With respect to the starting solutions, we see that the 1D virtual weights solution seems to perform a little better than Smith's rule. However, as we will see in Section 4.5, for larger instances this difference is much clearer.

4.4 Virtual weights heuristics

One way to get an IIA mechanism is by assigning each type a number and, for each type vector, scheduling the types in increasing order of their assigned numbers, breaking ties in some predetermined way. This results in an IIA mechanism since any two types will always get the same relative order. Duives et al. [19] show that, by assigning each type a virtual weight and ordering the types according to the ratio of virtual weight over processing time, the single-dimensional case is solved to optimality. This way of assigning virtual weights to the types also results in an IIA mechanism.

The reason that assigning virtual weights to the jobs results in optimal mechanisms in the single-dimensional case is that, for any Bayes-Nash implementable scheduling rule, the shortest paths in the type graph are the same. Therefore, the optimal mechanism can be derived from the virtual weights. In the two-dimensional case however, there are no such paths that are guaranteed to be shortest paths. Therefore the same method can not be applied to find the optimal mechanism. Still, there are several ways to apply the same idea in algorithms for the two-dimensional case. In this section we treat some experimental algorithms, that are based on the idea of virtual weights. The simplest of these algorithms is Algorithm 4.1. The idea is to start with the solution given by the 1D virtual weights and to iteratively improve the outcome by computing virtual weights over the shortest paths in the resulting type graph.

Algorithm 4.1 Virtual weight heuristic 1

```
    Input: Set $N$ of Jobs, per Job $j$ a set $T_j$ of types $t_j = (w_j, p_j)$ and $\varphi_j(\cdot)$
    Output: Order of all job types and payment per job type
    Compute 1D virtual weights
    while Condition holds do
5:      Compute virtual weights over shortest paths found
        Update virtual weights with ironed new virtual weights
        Compute expected start times
        Update arc lengths
        Get shortest paths in type graphs
10:     Payments = −length shortest paths
    end while
    Return: best solution in terms of total expected payment
```

After extensive computational experiments we found that this Algorithm 4.1 has the following problems.

1. There are, possibly, many different shortest paths in the type graph of one job.
2. Some shortest paths include arcs from higher to lower weight types. This results in lowering the virtual weight of the lower weighted job and possibly in negative virtual weights.

3. Highly variant behavior. When the new virtual weights are applied to the lengths of the arcs, many of the old shortest paths are not shortest paths anymore.

4. The algorithm produces solutions that sometimes have higher total payment than the starting solution. The algorithm does not improve the solution in every iteration.

5. The algorithm does not settle in one solution. Mostly, the algorithm tends to cycle between two or three solutions, until stopped artificially.

We conjecture that problem 4 is partly caused by problem 3. Still, we found that in certain cases the algorithm did not even find one solution that improved on the starting solution. The combination of problems 4 and 5 makes it hard to come up with a suitable iteration criterion, except for one based on the number of iterations or elapsed time. Because of these problems we implemented another heuristic in which we do the following.

1. We simply let a standard algorithm compute a single shortest path. [1]

2. If the shortest path includes any arcs from higher to lower weight types, we ignore their effect on the virtual weights.

3. Instead of using the new virtual weights instead of the old ones, we take a convex combination of both.

4. Without a suitable stop condition, we just let the algorithm do ten iterations and use the best result out of the ten.

Algorithm 4.2 shows the high level pseudo-code of this algorithm. In our tests the resulting algorithm performed exceptionally well.

4.5 Experiments

We did experiments to test the performance of the heuristics that where discussed in this chapter[2]. The experiments where done on random instances, in batches of a hundred, with a fixed number of jobs and for each job the same, fixed, number of types. The type space of each job was constructed randomly, such that $T_j = W_j \times P_j$, with W_j the set of possible weights for Job j and P_j the set of possible processing times for Job j. Figures 4.4 to 4.8 show plots of the results of these experiments and Table 4.3 shows the average computation times. We see that for instances where all jobs have a small type space, Algorithm 4.2 performs extremely well. For instances

[1]In our implementation we chose the implementation of the Bellman-Ford algorithm from the networkx package for Python [30].

[2]For these experiments we used an Intel(R) Core(TM)2 Duo E8400 3.00 GHz computer with Windows 7 64-bit operating system with 4.00 GB of memory. Using Python 3.2.5 for implementations and Gurobi 5.6.3 64-bit to solve the ILP and LP problems.

Algorithm 4.2 Virtual weight heuristic 2

```
    Input: Set $N$ of Jobs, per Job $j$ a set $T_j$ of types $t_j = (w_j, p_j)$ and $\varphi_j(\cdot)$
    Output: Order of all job types and payment per job type
    Compute 1D virtual weights
    for Iterations= $1, \dots, 10$ do
5:      Compute virtual weights over shortest paths found
        Update virt. weights with ironed (1/2(old virt. weights + new virt. weights))
        Compute expected start times
        Update arc lengths
        Get shortest paths in type graphs
10:     Payments = −length shortest paths
    end for
    Return: best solution in terms of total expected payment
```

where the type spaces are larger, it still performs about ten percent better than just applying the 1D virtual weights.

From Table 4.3 we see that, for the smaller instances, Algorithm 4.2 is a little slower than it is to solve the LP relaxation. However, note that the LP relaxation does not compute deterministic mechanisms. Even solving the smallest of these instances for the IIA ILP formulation was not possible on the machine that we used for these computations. Also, for larger instances, Algorithm 4.2 is much faster than solving the LP formulation. Furthermore, Figure 4.7 and Figure 4.8 seem to suggest that the quality of the solutions depends on the number of types per job, rather than the number of jobs or total number of types in the instance.

Instance size	4×50	9×50	16×50	25×50	25×100
LP relaxation	0.105	1.262	10.373	77.375	802.039
Smith's rule	0.036	0.162	0.520	1.369	4.671
1D virtual weights	0.038	0.168	0.532	1.375	4.615
Algorithm 4.2	0.411	1.851	5.871	15.399	50.849

Table 4.3: The average computation times in seconds for the results of Figures 4.4 to 4.8.

4.6 Discussion of the results

The heuristics for two-dimensional scheduling mechanism design, that we discussed in this chapter, are of an experimental nature. Although we can not prove any performance bounds, it is clear from the computational results that Algorithm 4.2 is very efficient for the computation of good deterministic IIA mechanisms for the two-dimensional scheduling mechanism design problem. It finds good solutions, even for instances that are way beyond the largest size that commercial solvers can reasonably

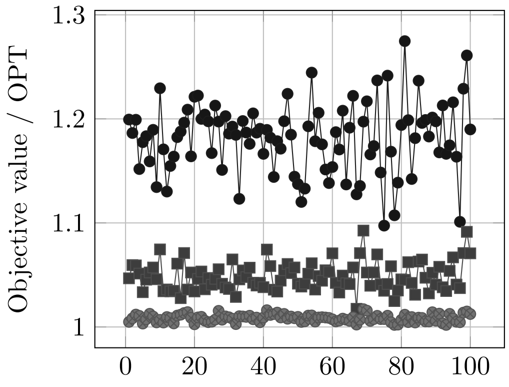


Figure 4.4: Results from a hundred random instances with 50 jobs, each with 4 types. The graph shows the objective value found for Smith's rule (upper), 1D virtual weights (middle) and Algorithm 4.2 (lower), all normalized to the optimal randomized objective.

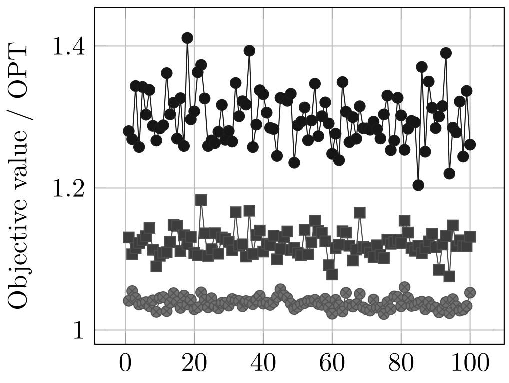


Figure 4.5: Results from a hundred random instances with 50 jobs, each with 9 types. The graph shows the objective value found for Smith's rule (upper), 1D virtual weights (middle) and Algorithm 4.2 (lower), all normalized to the optimal randomized objective.

solve the IIA ILP and in much less time than commercial solvers need to solve the randomized LP formulation. Also, it provides solutions that lie near the optimal randomized solution for instances with few types per job, while the solutions are still significantly better than the solutions provided by either Smith's rule or 1D virtual weights rule. We therefore think that our results are very encouraging towards future research on both fast and effective heuristics for multi-dimensional mechanism design problems.

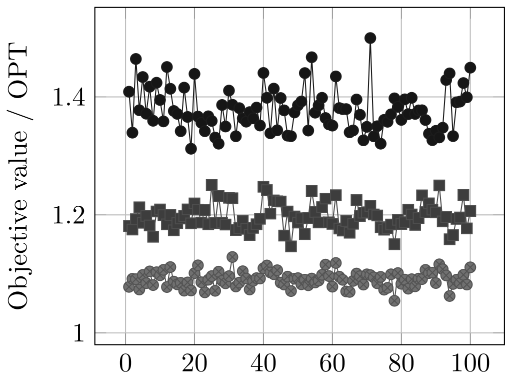


Figure 4.6: Results from a hundred random instances with 50 jobs, each with 16 types. The graph shows the objective value found for Smith's rule (upper), 1D virtual weights (middle) and Algorithm 4.2 (lower), all normalized to the optimal randomized objective.

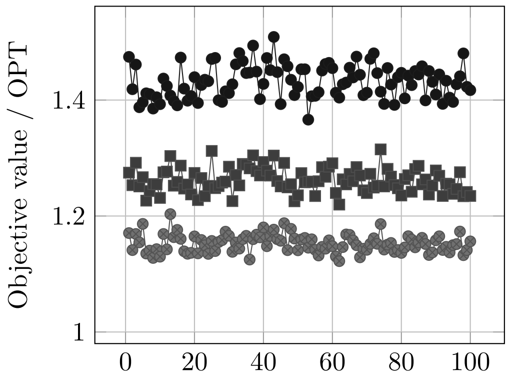


Figure 4.7: Results from a hundred random instances with 50 jobs, each with 25 types. The graph shows the objective value found for Smith's rule (upper), 1D virtual weights (middle) and Algorithm 4.2 (lower), all normalized to the optimal randomized objective.

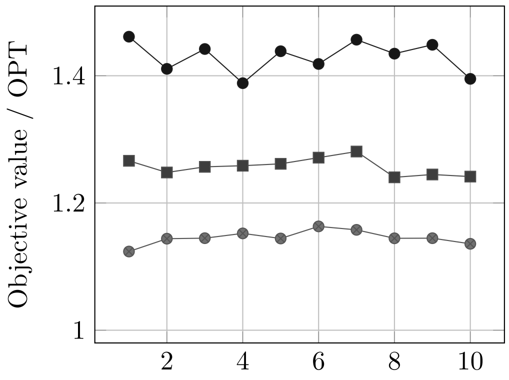


Figure 4.8: Results from ten random instances with 100 jobs, each with 25 types. The graph shows the objective value found for Smith's rule (upper), 1D virtual weights (middle) and Algorithm 4.2 (lower), all normalized to the optimal randomized objective.

CHAPTER 5

Intersecting and decomposing the scheduling polytope

In Chapter 3 we have seen that the two dimensional scheduling mechanism design problem can be solved with an LP relaxation. For each type vector, this results in a point that lies in the single machine scheduling polytope, as defined in Chapter 1. Recall that, for n jobs, the scheduling polytope is an $n-1$ dimensional polytope. Therefore, Carathéodory's theorem tells us that any such point can be written as a convex combination of at most n vertices of the polytope. In this chapter we investigate algorithms that, given a point in the scheduling polytope, find such a convex combination of vertices. We call such algorithms *decomposition algorithms*.

We know from Queyranne [58] that the separation problem for the scheduling polytope can be solved in time $\mathrm{O}(n \log n)$. From this and the ellipsoid method, it follows that a polynomial time decomposition algorithm exists [28]. Still, Cunningham [17] already remarked that it is interesting to find efficient combinatorial decomposition algorithms for specific polymatroids, and that it is, in general, not straightforward to do so, even if the underlying optimization problem is well understood and can be solved efficiently. Decomposition of feasible points into vertices also plays an important role in algorithms for submodular function minimization, starting with work by Cunningham [16, 17], and including the strongly polynomial time algorithms of Schrijver [65] and Iwata et al. [43].

For general polytopes, Grötschel et al. [29, Thm. 6.5.11] propose the following geometric approach to find a decomposition: Given some point x in a polytope P, pick an arbitrary vertex v of P, and compute the point $x' \in P$ where the half-line through v and x leaves P. This point lies on a face(t) of P, such that we can recur on that face(t). We call this the *GLS method*. One iteration of the GLS method is illustrated in Figure 5.1.

To efficiently use the GLS method, we need a way to efficiently compute x' and a facet on which it lies. For polymatroids, this can be done with an algorithm described by Fonlupt and Skoda [24]. For the scheduling polytope, a direct application of their result leads to an algorithm that runs in time $\mathrm{O}(n^8)$.

In this chapter, we propose two algorithms. The first algorithm computes, in time $\mathrm{O}(n^2 \log n)$, the intersection of a line with the scheduling polytope. This algorithm, together with GLS method, results in an algorithm that decomposes any point in the scheduling polytope into at most n vertices in time $\mathrm{O}(n^3 \log n)$. The second

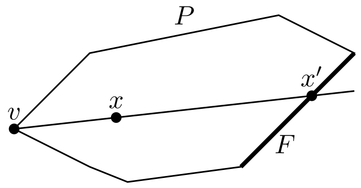


Figure 5.1: Illustration of the decomposition algorithm by Grötschel, Lovász, and Schrijver. From some vertex $v \in P$, extend a half-line from v in the direction $x - v$, until it intersects a lower dimensional face F of P in a point x'. The point x can be written as a convex combination of v and x'. Recurs with the face F and the intersection point x' to obtain a convex combination of vertices of F that yields x'.

algorithm is a decomposition algorithm that takes advantage of the fact that the scheduling polytope is a zonotope, which we define later. It is an algorithm that decomposes a point in the scheduling polytope in time $\mathrm{O}(n^2)$. The algorithm applies the GLS method to an appropriately chosen subpolytope of the scheduling polytope. The resulting decomposition into vertices of that subpolytope can then easily be translated into a decomposition into vertices of the scheduling polytope itself. In fact, it turns out that this algorithm is a generalization of a decomposition algorithm for the permutahedron by Yasutake et al. [69]. Therefore, it provides a geometric interpretation of their algorithm.

We consider the scheduling polytope as described by (1.2) and (1.3). Recall that, if $p_j > 0$ for all $j \in N$, none of these inequalities is redundant, and the dimension is $n - 1$ [58]. For all algorithmic purposes that we can think of, the degenerate case, where for some jobs $p_j = 0$, does not add anything interesting, since we can simply eliminate such jobs and reintroduce them afterwards. In particular, this is true for the problems we address here. Thus, we assume that $p_j > 0$ for all jobs $j \in N$ from now on.

First we state a lemma that directly follows from Queyranne [58]; it shows that the separation problem for the scheduling polytope can in fact be solved by sorting. Let Q^S denote the scheduling polytope for start time vectors and let $g(K)$ be defined by (1.1).

Lemma 5.1. *Let S be a given vector sorted such that $S_1 \leq S_2 \leq \ldots \leq S_n$. Then $S \in Q^S$ if and only if $\sum_{j\in N} p_j S_j = g(N) - (1/2)\sum_{j\in N}(p_j)^2$ and $\sum_{j\in K} p_j S_j \geq g(K) - (1/2)\sum_{j\in K}(p_j)^2$ holds for all $K = \{1, \ldots, k\}$, $k \in N$. In particular, if there is a set $K \subseteq N$ that violates* (3.7)*, then there is a $k \in N$ such that the set $\{1, \ldots, k\}$ also violates* (3.7).

Proof. Let

$$\Gamma(K) := g(K) - \frac{1}{2}\sum_{j\in K}(p_j)^2 - \sum_{j\in K} p_j S_j \ , \tag{5.1}$$

be the function that measures the violation of (3.7). Queyranne [58, Lem. 5.2] shows that for given S, if $K \subseteq N$ maximizes the violation $\Gamma(K)$ then $l \notin K$ if and only if $S_l \geq \sum_{j \in K} p_j$. Suppose K is a set that maximizes $\Gamma(K)$. Choose k such that $S_k < \sum_{j \in K} p_j$ and $S_{k+1} \geq \sum_{j \in K} p_j$. Then $j \in K$ for all $j \in \{1, \ldots, k\}$ and $j \notin K$ for all $j \in \{k+1, \ldots, n\}$, so $K = \{1, \ldots, k\}$. Therefore if there is a set that violates (3.7), i.e. $\Gamma(K) > 0$, then there is an index k such that the set $\{1, \ldots, k\}$ maximizes that violation and thus also violates (3.7). □

5.1 Intersection of a line and the scheduling polytope

In this section, we describe an algorithm that computes the point where a line, given by $L = \{x + \lambda y \mid \lambda \in \mathbb{R}\}$, intersects the scheduling polytope Q^S. Let $\ell(\lambda) := x + \lambda y$ and assume $y \neq 0$. The simple idea is this: For each point, $\ell(\lambda)$, on the line L, we consider the non-increasing order of its components. Every such order, σ, induces n nested sets, $\{\sigma(1), \ldots, \sigma(k)\}$, for each $k = 1, \ldots, n$, which correspond to the sets K from Lemma 5.1. Here $\sigma(i)$ denotes the component that is i-th in the order σ. The line L intersects a half-space, corresponding to (3.7), in a point where $\Gamma(K) = 0$ for $\ell(\lambda)$ and K is a nested set. Now, we enumerate all orders on L and their induced nested sets. From these we get two points that are the intersections of L with one of the faces of the scheduling polytope, while all points of L, in between, are inside the polytope. This results in an efficient algorithm, as all points on L, have, in total, at most $O(n^2)$ induced orders σ of the components of $\ell(\lambda)$.

Lemma 5.2. *The vectors $\ell(\lambda)$ on the line L have at most $O(n^2)$ different induced orders σ of their components.*

Proof. On the line L, the relative order of components i and j of vector $\ell(\lambda)$ can change at most once, since L is a line. □

By Lemma 5.2 we have no more than $O(n^2)$ induced orders on L and it is not hard to see that they can all be computed in $O(n^3)$ total time. These $O(n^2)$ orders give rise to no more than $O(n^3)$ candidate sets K for the intersection of half-spaces, (3.7), with the line L. We can compute the intersection of the half-space induced by K with L in time $O(n)$, by solving $\Gamma(K) = 0$. Hence, for any two candidates we can decide, in time $O(n)$, which of the two intersections is closer to Q^S. By this line of arguments we get an $O(n^4)$ time bound for computing a facet on which the half-line L leaves polytope Q^S. A slightly more clever bookkeeping, however, allows to obtain a better computation time.

Theorem 5.3. *Let Q^S be the scheduling polytope for start times, induced by the vector of processing times $p \in \mathbb{R}^n_{>0}$. For given $x, y \in \mathbb{R}^n$, $y \neq 0$, the computation of the intersection of the line $L = \{x + \lambda y \mid \lambda \in \mathbb{R}\}$ with Q^S, together with the facets of Q^S, on which L intersects, can be done in time $O(n^2 \log n)$.*

Proof. The idea of the algorithm is as follows. The relative order of any pair of components i and j, of $\ell(\lambda) \in L$, can change at most once on the line L. For each such pair i, j, such that $y_i \neq y_j$, this order changes exactly when the components have equal value, i.e. at the point $\ell(\lambda(i,j))$, where

$$\lambda(i,j) = \frac{x_i - x_j}{y_j - y_i} \quad .$$

For any pair i, j with $y_i = y_j$ the relative order of i and j is the same over the whole line L, so these need not be considered.

These points, $\ell(\lambda(i,j))$, divide L into intervals, I, on which there is a single induced order σ of the components of the vectors $\ell \in I$. This not only bounds the number of induced orders by $\mathrm{O}(n^2)$, it also bounds the total number of distinct nested sets $K = \{\sigma(1), \dots \sigma(k)\}$, $k = 1, \dots, n$, for all induced orders σ, by $\mathrm{O}(n^2)$. This because, at $\ell(\lambda(i,j))$, only the relative order of the components i and j changes[1]. This means that i and j are consecutive in the induced orders of the two intervals incident with the point $\ell(\lambda(i,j))$. Therefore, all induced nested sets K for these two intervals are identical, except for the one which contains exactly one of i and j.

Each of these induced nested sets K gives rise to one inequality (3.7) and for each such set K, for which $\sum_{j \in K} p_j y_j \neq 0$, the line L intersects the hyper-plane defined by (3.7) for K. Let us denote by $\delta(K)$, the parameter such that $\ell(\delta(K))$ is exactly this intersection point. Note that $\delta(K)$ can be easily computed if L and K are given. If $\sum_{j \in K} p_j y_j = 0$, then the line, L, and the hyper-plane, induced by (3.7) for K, are affinely dependent, and there is no intersection. The values $\delta(K)$ can now be divided into two sets: those for which $q \geq \delta(K)$ for any $\ell(q) \in L \cap Q^S$ and those for which $q \leq \delta(K)$ for any $\ell(q) \in L \cap Q^S$. These provide lower bounds and upper bounds for the intersection of L and Q^S, respectively. The largest lower bound, denoted $\underline{\delta}$, and the smallest upper bound, denoted $\overline{\delta}$, exactly yield the intersection of L and Q^S, $\{x + \lambda y \mid \lambda \in [\underline{\delta}, \overline{\delta}]\}$.

To see why, note that for every $\ell(\lambda)$, with $\underline{\delta} \leq \lambda \leq \overline{\delta}$, (3.7) holds for all K that are induced nested sets of vector $\ell(\lambda)$. Therefore, we have from Lemma 5.1 that $\ell(\lambda) \in Q^S$, and $\ell(\lambda) \in L$ by definition. Also, for any $\ell(\lambda)$ with $\lambda > \overline{\delta} = \delta(K)$ for some nested set K, (3.7) is violated for K and thus $\ell(\lambda) \notin Q^S$. Likewise for any $\ell(\lambda)$ with $\lambda < \underline{\delta}$, we have $\ell(\lambda) \notin Q^S$.

The idea is now to compute $\overline{\delta}$ and $\underline{\delta}$, together with the corresponding facet inducing nested sets, K, by 'moving' along L in the order of sorted values $\lambda(i,j)$, and updating the nested sets K, and all other necessary parameters, incrementally. Algorithm 5.1 gives the complete pseudocode for computing the intersection of a line L with the scheduling polytope Q^S.

We briefly explain the pseudocode and analyze the computation time in the following. In line 3, the values $\underline{\delta}$ and $\overline{\delta}$ and $\mathcal{L}$ and $\mathcal{K}$, are initialized. $\mathcal{L}$ is the list of parameters $\lambda(i,j)$, on which the induced orders of $\ell(\lambda)$ change. $\mathcal{K}$ contains all necessary information needed for the nested sets K and incremental updating in the course of the algorithm. The computation time of this initialization is $\mathrm{O}(1)$.

[1]If multiple pairs of components change their relative order in the same point $\ell(\lambda)$, we can treat these separately in the analysis to obtain the same result.

Algorithm 5.1 Intersection algorithm

```
    Input : processing time vector $p \in \mathbb{R}^n_{>0}$, vectors $x \in \mathbb{R}^n$ and $0 \neq y \in \mathbb{R}^n$
    Output: values $\underline{\delta}$ and $\overline{\delta}$
    $\underline{\delta} := -\infty$, $\overline{\delta} := \infty$, $\mathcal{L} := [\,]$, $\mathcal{K} := [\,]$.
    for $i = 1$ to $n$ do
 5:     for $j = 1$ to $i - 1$ and $y_j \neq y_i$ do
            $\lambda(i,j) := (x_i - x_j)/(y_j - y_i)$
            $\mathcal{L} := \mathcal{L} + [((i,j), \lambda(i,j))]$
        end for
    end for
10: Sort $\mathcal{L}$ increasing in $\lambda(i,j)$
    $\lambda^0 := \lambda(i,j)$ of first element of $\mathcal{L}$
    $\sigma :=$ order of $\ell(\lambda^0 - 1)$
    for $j = 1$ to $n$ do
        $K(j) := \{\sigma(1), \ldots, \sigma(j)\}$
15:     $P(K(j)) := P(K(j-1)) + p_j$
        $F(K(j)) := F(K(j-1)) + P(K(J-1))p_j - p_j x_j$
        $Y(K(j)) := Y(K(j-1)) + p_j y_j$
        $\mathcal{K} := \mathcal{K} + [(K(j), P(K(j)), F(K(j)), Y(K(j)))]$
    end for
20: for $K \in \mathcal{K}$ and $Y(K) \neq 0$ do
        $\delta(K) := \frac{F(K)}{Y(K)}$
        if $F(K) - (\delta(K) + 1)Y(K) > 0$ then
            $\overline{\delta} := \min\{\overline{\delta}, \delta(K)\}$
        else
25:         $\underline{\delta} := \max\{\underline{\delta}, \delta(K)\}$
        end if
    end for
    for $(\lambda(i,j), (i,j)) \in \mathcal{L}$ do
        if $\sigma^{-1}(i) < \sigma^{-1}(j)$: then
30:         $K := \sigma^{-1}(i)$-th element in $\mathcal{K}$
            $K' := K \setminus \{i\} \cup \{j\}$
            $P(K') := P(K) - p_i + p_j$
            $F(K') := F(K) - (\frac{1}{2}P(K) - \frac{1}{2}p_i^2 - p_i x_i) + (\frac{1}{2}P(K') - \frac{1}{2}p_j^2 - p_j x_j)$
            $Y(K') := Y(K) - p_i y_i + p_j y_j$
35:     else
            $K := \sigma^{-1}(j)$-th element in $\mathcal{K}$
            $K' := K \setminus \{j\} \cup \{i\}$
            $P(K') := P(K) - p_j + p_i$
            $F(K') := F(K) - (\frac{1}{2}P(K) - \frac{1}{2}p_j^2 - p_j x_j) + (\frac{1}{2}P(K') - \frac{1}{2}p_i^2 - p_i x_i)$
40:         $Y(K') := Y(K) - p_j y_j + p_i y_i$
        end if
        Replace $(K, P(K), F(K), Y(K))$ in $\mathcal{K}$ by $(K', P(K'), F(K'), Y(K'))$
        Switch $i$ and $j$ in the order $\sigma$
        $\delta(K') := \frac{F(K')}{Y(K')}$
45:     if $F(K') - (\delta(K') + 1)Y(K') > 0$ then
            $\overline{\delta} := \min\{\overline{\delta}, \delta(K')\}$
        else
            $\underline{\delta} := \max\{\underline{\delta}, \delta(K')\}$
        end if
50: end for
    return $\underline{\delta}, \overline{\delta}$
```

In lines 4 to 9, the values $\lambda(i,j)$ are computed and added to $\mathcal{L}$ and in line 10, $\mathcal{L}$ is sorted in ascending order. Since there are at most $O(n^2)$ of these values, the sorting can be done in time $O(n^2 \log n)$.

In line 11, λ^0 is set to the smallest $\lambda(i,j)$ and in line 12, σ is set to be the order of $\ell(\lambda^0 - 1)$. This order corresponds to the ascending order of all components of

$\ell(\lambda)$, for $\lambda < \lambda^0$. It can be computed in time $O(n \log n)$.

In lines 13 to 19, all nested subsets, $K(j)$, that are induced by σ are stored in $\mathcal{K}$ together with the values $P(K(j))$, $F(K(j))$ and $Y(K(j))$. Note that $F(K(j)) - \lambda Y(K(j))$ is exactly equal to $\Gamma(K(j))$ for the point $\ell(\lambda)$. Therefore $\Gamma(K(j)) = 0$ for $\ell(\delta(K(j)))$. Computing the values $P(K(j))$, $F(K(j))$ and $Y(K(j))$ is done incrementally in time $O(1)$. Since there are at most $O(n)$ nested subsets $K(j)$, computing all of them can be done in time $O(n)$.

In line 21, $\delta(K)$ is computed, such that $\Gamma(K) = 0$ for $\ell(\delta(K))$, and the if-clause, on line 22, determines if (3.7) for K is satisfied by points $\ell(\lambda)$, for $\lambda > \delta(K)$, or by points $\ell(\lambda)$, for $\lambda < \delta(K)$. In case of the former, $\delta(K)$ is an upper bound and, in case of the latter, $\delta(K)$ is a lower bound, which is then updated accordingly in lines 23 or 25. All steps can be computed in time $O(1)$, there are $O(1)$ computations per subset K and there are $O(n)$ subsets, therefore all computations can be done in time $O(n)$.

In lines 28 to 50, for each $\lambda(i,j)$, in ascending order, we first determine how the order will change, i.e., whether i was before j or the other way around. Let us assume the former, the latter is symmetric. Then, K is the $\sigma^{-1}(i)$-th induced subset of σ, i.e. the subset containing i but not j. Here $\sigma^{-1}(i)$ denotes the position that component i has in order σ. K' is computed as the new induced subset and $P(K')$, $F(K')$ and $Y(K')$ as the corresponding values. These replace $(K, P(K), F(K), Y(K))$ in $\mathcal{K}$, while i and j are switched in σ. Then, the value $\delta(K')$ is computed, it is again determined if the upper or the lower bound has to be updated and this is done accordingly. Each step can be computed in time $O(1)$ and there are $O(1)$ computations per iteration. There are $O(n^2)$ iterations, therefore all computation can be done in time $O(n^2)$.

If the returned values $\overline{\delta} < \underline{\delta}$, then the intersection of Q^S and L is empty. Otherwise the interval, $\{x + \lambda y \mid \lambda \in [\underline{\delta}, \overline{\delta}]\}$, is the intersection of Q^S and L.

The computation time of the algorithm is dominated by the sorting of the values $\lambda(i,j)$ in line 10. So the total computation time of the algorithm is $O(n^2 \log n)$. We can easily find the facets at which the line L and the scheduling polytope Q^S intersect by repeating lines 13 to 19 for $\overline{\delta}$ and $\underline{\delta}$. □

5.2 Decomposition algorithm

If we combine Algorithm 5.1 with the GLS method we obtain a decomposition algorithm for the scheduling polytope that runs in time $O(n^3 \log n)$. In this section we show that this can be improved to time $O(n^2)$ by using a geometric algorithm, based on the fact that the scheduling polytope is a zonotope. In contrast to Algorithm 5.1, the next algorithm, while it finds a decomposition faster, it does not yield the intersection of the scheduling polytope with a line.

Essentially, we show two things. First, we show that there is an $O(n^2)$ decomposition algorithm for the single machine scheduling polytope. The core of our algorithm remains the GLS method. However, we apply the method to a specific subpolytope of a polyhedral subdivision of the scheduling polytope for *half times*, Q. We obtain

Q by shifting Q^S by half the processing times of the jobs: $Q = Q^S + p/2$. Second, it turns out that our algorithm is a generalization of a decomposition algorithm for the permutahedron, by Yasutake et al. [69]. Our algorithm augments their result with a simple, geometric interpretation. In particular, this shows that their algorithm is in fact, also, an implementation of the GLS method.

It should be mentioned that the idea of using half times, also referred to as midpoints, is not new in scheduling. It has proven to be helpful particularly for the design and analysis of approximation algorithms. Phillips et al. [57] were probably the first to use half times to analyze an approximation algorithm, and Munier et al. [48] were the first to use half times explicitly in the design of approximation algorithms.

Crucial to get our results is exploitation of the fact that the scheduling polytope is a zonotope. This means that all its faces are centrally symmetric. As each of the centers of a given face has a representation by two vertices, it suffices to decompose a given point into (certain) centers. To decompose a given point into centers, we consider the polyhedral subdivision of the scheduling polytope that is induced by these centers. This is also called a barycentric subdivision [46]. We can show that, for the scheduling polytope for half times, this subdivision has a simple, linear, description, which we can exploit algorithmically.

We believe that our results are interesting due to the following reasons. First, consider applying the GLS method directly to the scheduling polytope. In order to obtain an $\mathrm{O}(n^2)$ implementation, one would have to compute a face F and the intersection point of the halfline through v and x with F in $\mathrm{O}(n)$ time in each iteration. We do not see how to do this. Second, consider a naive, unit-cost, encoding of the output. Then, the $\mathrm{O}(n^2)$ implementation is only linear in the output size. Third, our structural results shed new light on a well-studied object in polyhedral combinatorics, namely the single machine scheduling polytope.

Since the permutahedron and the scheduling polytope are so similar, an affine transformation from one to the other even exists, one may ask why we do not simply transform the scheduling polytope to the permutahedron, then decompose it and transform it back. We answer this question with the following lemma which shows that such a transformation does not preserve decompositions.

Lemma 5.4. *Transforming different convex combinations of vertices of the permutahedron, that describe the same point, to the scheduling polytope may result in convex combinations of vertices of the scheduling polytope, that describe different points.*

Proof. Consider the permutahedron for $(1, 2, 3)$. We have

$$\frac{1}{3}(1,2,3) + \frac{1}{3}(3,1,2) + \frac{1}{3}(2,3,1) = (2,2,2) \ , \tag{5.2}$$

$$\frac{1}{2}(1,2,3) + \frac{1}{2}(3,2,1) = (2,2,2) \ . \tag{5.3}$$

However, if we translate the orders to their corresponding completion time vectors in the scheduling polytope with $p = (1, 2, 3)$, we get

$$\frac{1}{3}(1,3,6) + \frac{1}{3}(6,3,5) + \frac{1}{3}(4,6,3) = (\frac{11}{3}, \frac{11}{3}, \frac{14}{3}) \ , \tag{5.4}$$

$$\frac{1}{2}(1,3,6) + \frac{1}{2}(6,5,3) = (\frac{7}{2}, \frac{8}{2}, \frac{9}{2}) \ . \tag{5.5}$$

□

5.2.1 Zonotopes

Definition 5.1 (centrally symmetric polytope, zonotope)**.** Let $P \subseteq \mathbb{R}^n$ be a polytope.

P is *centrally symmetric* if it has a center $c \in P$, such that $c + x \in P$ if and only if $c - x \in P$.

If all faces of P are centrally symmetric, then P is called a *zonotope*.

An equivalent definition of centrally symmetric is that there exists a center, $c \in P$, such that for all $x \in P$ also $2c - x \in P$.

Also zonotopes have alternative definitions. They are exactly the images of (higher-dimensional) hypercubes under affine projections, and they are exactly the Minkowski sum of line segments [71]. The standard textbook example for zonotopes is, actually, the permutahedron [71].

The scheduling polytope with arbitrary processing times is a zonotope, too. This can be seen in several ways. For example, as we have seen in Lemma 3.1, the scheduling polytope can be obtained as an affine transformation from a hypercube in dimension $\binom{n}{2}$, via linear ordering variables [59, Thm. 4.1].

Theorem 5.5 (Queyranne and Schulz [59, Thm. 4.1])**.** *The scheduling polytope is a zonotope.*

With respect to the centers of the faces of the scheduling polytope for halftimes, we have the following lemma.

Lemma 5.6. *Consider an arbitrary face F of Q, defined by the ordered partition $(D_1, \ldots, D_k)$, then the center of symmetry (or barycenter or center of mass), $c(F)$, of F is given by*

$$c(F)_j = \sum_{\ell=1}^{i-1} \sum_{h \in D_\ell} p_h + \frac{1}{2} \sum_{h \in D_i} p_h \quad \textit{for all} \quad j \in D_i \ . \tag{5.6}$$

Given that a face F of Q corresponds to some ordered partition $(D_1, \ldots, D_k)$, this is not difficult to verify. For the sake of completeness, we give a proof here.

Proof. Let F be any face of Q, and let v be a vertex of F. Let $(D_1, \ldots, D_k)$ be the ordered partition corresponding to F. Then v corresponds to an ordering such that jobs in D_a are ordered before jobs in D_b for all $a < b$. Now let v' be the vertex of F

that corresponds to the following order: for any two jobs $i, j \in D_a$ and $i \neq j$, we let j be ordered before i if and only if i is ordered before j in v. Note that, for any v, there is exactly one v' in F that satisfies this order.

We show that indeed $c(F) = \frac{1}{2}(v + v')$. Suppose that $j \in D_a$. Then for any $b < a$, in both v and v' any job $i \in D_b$ is ordered before job j. For any $b > a$, in both v and v' any job $i \in D_b$ is ordered after job j. And, for any job $i \in D_a$, $i \neq j$, job i is ordered before job j in one of v and v' and ordered after job j in the other. From this we have that

$$\frac{1}{2}(v + v')_j = \sum_{\ell=1}^{a-1} \sum_{h \in D_\ell} p_h + \frac{1}{2} \sum_{h \in D_a} p_h \quad \text{for all} \quad j \in D_a \ ,$$

which is indeed $c(F)_j$ as defined by Lemma 5.6. Therefore we have that $c(F) = \frac{1}{2}(v + v')$ and thus $v' = 2c(F) - v$. Since this holds for any vertex $v \in f$, it follows that for any point $x \in f$ there exists a point $x' \in f$ such that $x' = 2c(F) - x$ and thus $c(F)$ is the center of F. □

In particular, observe that all $j \in D_i$ have the same value, $c(F)_j$, and the center of Q is the point c where all values c_i coincide, i.e., $c_1 = \ldots = c_n$. Note that this is no longer true if we consider the scheduling polytope for start or completion times. The property that all faces of a zonotope are centrally symmetric and the simple description of these centers by Lemma 5.6, is important for the design of the decomposition algorithm in Section 5.2.3.

5.2.2 Barycentric subdivision

Consider the following, polyhedral subdivision of the scheduling polytope Q. For any vertex v of Q, define polytope Q_v^c as the convex hull of all barycenters $c(F)$ of faces F that contain v:

$$Q_v^c := \operatorname{conv}\{c(F) \mid v \in f\} \ .$$

Then we have $Q = \bigcup_v Q_v^c$. By construction, v is the only vertex of Q that is also a vertex of Q_v^c. The subdivision thus obtained is also known as *barycentric subdivision* [46].

Another polyhedral subdivision of the scheduling polytope Q is obtained by subdividing the polytope according to orders as follows.

Definition 5.2. Let $P \subseteq \mathbb{R}^n$ be a polytope. We define a relation $\sim$ on P as follows: for two points $x, y \in P$, we have $x \sim y$ if there exists a permutation $\sigma : \{1, \ldots, n\} \to \{1, \ldots, n\}$ such that both $x_{\sigma(1)} \leq \ldots \leq x_{\sigma(n)}$ and $y_{\sigma(1)} \leq \ldots \leq y_{\sigma(n)}$.

Based on this definition, define for any vertex $v \in Q$ the polytope

$$Q_v^\sigma := \{x \in Q \mid x \sim v\} \ .$$

Because every permutation σ is represented by a vertex of Q, we have $Q = \bigcup_v Q_v^\sigma$, and v is the only vertex of Q that is also a vertex of Q_v^σ.

The following two lemmas encode the core and geometric intuition behind the decomposition algorithm that we develop in Section 5.2.3. They show that the two polyhedral subdivisions above are in fact equivalent. Thus, we obtain a description of the barycentric subdivision in terms of vertices and facets, all of which can be described explicitly by simple expressions. These insights can be exploited algorithmically.

Lemma 5.7. *Let Q be the single machine scheduling polytope for half times, let v be an arbitrary vertex of Q and let σ denote a permutation such that $v_{\sigma(1)} \leq \ldots \leq v_{\sigma(n)}$. Then Q_v^σ has the following, linear description:*

$$H_{\sigma(j)} \leq H_{\sigma(j+1)} \qquad \text{for all } j \in \{1,\ldots,n-1\} \ , \tag{5.7}$$

$$\sum_{i=1}^{j} H_{\sigma(i)} p_{\sigma(i)} \geq \frac{1}{2}\left(\sum_{i=1}^{j} p_{\sigma(i)}\right)^2 \qquad \text{for all } j \in \{1,\ldots,n-1\} \ , \text{ and} \tag{5.8}$$

$$\sum_{j\in N} H_j p_j = \frac{1}{2}\left(\sum_{j\in N} p_j\right)^2 \quad . \tag{5.9}$$

Proof. $Q_v^\sigma \subseteq Q$, (5.8) and (5.9) are satisfied for every point in Q_v^σ. Since σ is the only permutation with $v_{\sigma(1)} \leq \ldots \leq v_{\sigma(n)}$, we have that H satisfies (5.7) if $H \sim v$. Therefore, (5.7) holds for any point in Q_v^σ.

It remains to be shown that (5.7), (5.8), and (5.9) imply $H \in Q_v^\sigma$. Let H satisfy (5.7), (5.8) and (5.9). For simplicity of notation and without loss of generality, let all vectors be sorted such that $H_i \leq H_j$ if and only if $i \leq j$. Then, for each j, we have

$$\left(\sum_{i=1}^{j} p_i\right) H_j \geq \sum_{i=1}^{j} p_i H_i \geq \frac{1}{2}\left(\sum_{i=1}^{j} p_i\right)^2 \quad .$$

Thus, $H_j \geq \frac{1}{2}\sum_{i=1}^{j} p_i$ for all j. Now suppose H satisfies (5.7), (5.8), and (5.9), but $H \notin Q$. Then there is a set K of minimal cardinality, such that (1.2) is not satisfied. This means that

$$\sum_{i\in K} p_i H_i < \frac{1}{2}\left(\sum_{i\in K} p_i\right)^2 \quad .$$

But then, for $j = \max_{k\in K} k$, we have

$$\begin{aligned}
\sum_{i\in K\setminus\{j\}} p_i H_i &= \sum_{i\in K} p_i H_i - p_j H_j < \frac{1}{2}\left(\sum_{i\in K} p_i\right)^2 - p_j H_j \\
&\leq \frac{1}{2}\left(\sum_{i\in K} p_i\right)^2 - p_j \frac{1}{2}\left(\sum_{i=1}^{j} p_i\right) \\
&\leq \frac{1}{2}\left(\sum_{i\in K} p_i\right)^2 - p_j \frac{1}{2}\left(\sum_{i\in K} p_i\right) = \frac{1}{2}\left(\sum_{i\in K\setminus\{j\}} p_i\right)^2 \quad .
\end{aligned}$$

This contradicts that K is a set of minimal cardinality that does not satisfy (1.2). So (5.7), (5.8), and (5.9) imply $H \in Q$.

Now suppose $H \in Q \setminus Q_v^\sigma$, then $H \in Q_{v'}^\sigma$ for some other vertex $v' \in Q$, which would imply that (5.7) is not valid for H. Hence, $H \in Q_v^\sigma$. $\square$

Lemma 5.8. *Let Q be the single machine scheduling polytope in half times. Then, for all vertices v of Q, we have*

$$Q_v^c = Q_v^\sigma \ .$$

Proof. Lemma 5.6 implies that the vertices of Q_v^c are given by (5.6) for all $F \ni v$. From (5.6), we have $q \sim v$ for any vertex q of Q_v^c. It follows that $Q_v^c \subseteq Q_v^\sigma$.

Now, by Lemma 5.7, any vertex of Q_v^σ is obtained by having $n-1$ tight constraints among (5.7) and (5.8). Consider any such vertex q of Q_v^σ.

Let $\ell \in \{1, \ldots, n-1\}$. If (5.8) is tight for q for $k = \ell$, then (5.7) cannot be tight for q for $j = \ell$. This is because if (5.8) is tight for q and $k = \ell$, then jobs $1, \ldots, \ell$ are scheduled before jobs $\ell + 1, \ldots, n$. Therefore,

$$q_{\ell+1} \geq \frac{1}{2} p_{\ell+1} + \sum_{j=1}^{\ell} p_j$$

and

$$q_\ell \leq \frac{1}{2} p_\ell + \sum_{j=1}^{\ell-1} p_j \ .$$

Thus, $q_\ell < q_{\ell+1}$ since all processing times are assumed to be positive. This implies that for any $\ell \in \{1, \ldots, n-1\}$, we have that q satisfies exactly one of the following: (5.8) is tight for $k = \ell$ or (5.7) tight for $j = \ell$. The inequalities (5.8) that are tight for q induce an ordered partition $(D_1, \ldots, D_k)$ that corresponds to a face $F \ni v$. The inequalities (5.7) that are tight for q ensure that $q_j = q_{j+1}$ for all $j \in D_i$ and any $i \in \{1, \ldots, k\}$.

It follows that $q = c(F)$ and, thus, q is a vertex of Q_v^c. Since this holds for any vertex of Q_v^σ, we have $Q_v^\sigma \subseteq Q_v^c$. Thus, $Q_v^\sigma = Q_v^c$. $\square$

For simplicity of notation, we define $Q_v := Q_v^c$ $(= Q_v^\sigma)$.

Figure 5.2 illustrates the barycentric subdivision of the scheduling polytope. It shows the scheduling polytope for three jobs together with its barycentric subdivision (indicated by dashed lines). The subpolytope containing vertex v_{213} contains all vectors $H \in Q$ for which $H_2 \leq H_1 \leq H_3$. Its vertices are v_{213}, and all centers of faces on which v_{213} lies. Its facets are defined by $H_1 p_1 + H_2 p_2 + H_3 p_3 = (p_1 + p_2 + p_3)^2$ together with one of the following equalities:

$$\begin{aligned} H_1 p_1 + H_2 p_2 &= (p_1 + p_2)^2 \ , \\ H_2 p_2 &= (p_2)^2 \ , \\ H_2 &= H_1 \ , \\ H_3 &= H_1 \ . \end{aligned}$$

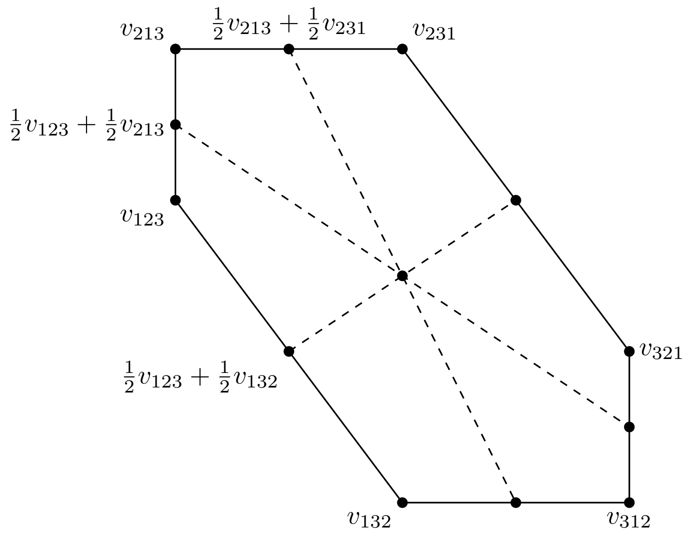

v_{213}
$\frac{1}{2}v_{213} + \frac{1}{2}v_{231}$
v_{231}
$\frac{1}{2}v_{123} + \frac{1}{2}v_{213}$
v_{123}
v_{321}
$\frac{1}{2}v_{123} + \frac{1}{2}v_{132}$
v_{132}
v_{312}

Figure 5.2: Barycentric subdivision of a scheduling polytope with three jobs. v_{ijk} denotes the vertex corresponding to the order i, j, k

5.2.3 Decomposition on a subpolytope

Based on Lemma 5.7, we next develop a decomposition algorithm for the scheduling polytope that runs in time $O(n^2)$. This algorithm can be seen as a generalization of a decomposition algorithm for the permutahedron, by Yasutake et al. [69]. We argue here that this algorithm is in fact an application of the GLS method [29, Thm. 6.5.11].

We first describe the high level idea of the decomposition algorithm for the scheduling polytope, before diving into the technical details.

We know that any point $H \in Q$ lies in a subpolytope Q_v of the barycentric subdivision of Q, namely for a vertex v for which $v \sim h$ according to Definition 5.2.[2] Moreover, Q_v is described by inequalities (5.7) and (5.8), and the vertices of Q_v consist of the points $\frac{v+v'}{2}$ for all vertices v' of Q. This means that a decomposition of H into vertices of Q_v also yields a decomposition into vertices of Q.

The idea of our algorithm is as follows: We find a decomposition of H into vertices of Q_v by using the GLS method [29, Thm. 6.5.11]. The idea of this algorithm is illustrated in Figure 5.3: Given $H = H^1 \in Q_v$ (we have $v = v^1$), we extend the difference vector $H^1 - v^1$ towards the intersection with a lower dimensional face of Q_v (this will be a facet of Q_v, unless we accidentally hit a face of even lower dimension). Then recurse with this intersection point and the face on which it lies. To arrive at the claimed computation time, it is crucial that both the intersection point and

[2]In case of ties, H lies on the intersection of several of such subpolytopes, namely those corresponding to vertices v with $v \sim h$. We can break such ties arbitrarily.

the face(t) on which it lies can be computed in time $O(n)$. This is indeed possible because of Lemma 5.7. As the number of iterations is bounded by the dimension of Q_v, which is equal to the dimension of Q, this gives an $O(n^2)$ implementation. Finally, by the fact that all vertices of Q_v can be written as $\frac{v+v'}{2}$ for vertices v' of Q, we obtain a decomposition of H into at most n vertices of Q.

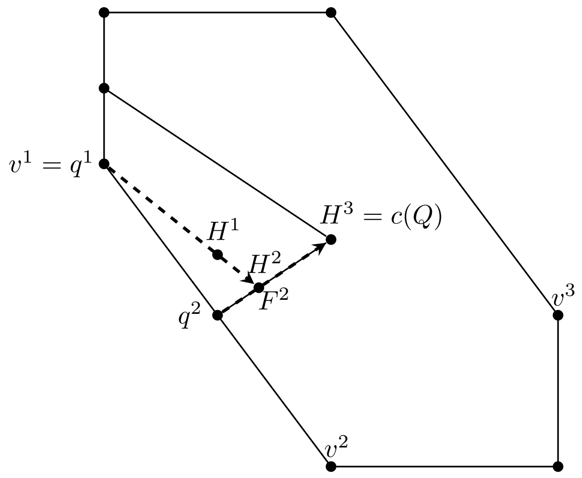


Figure 5.3: Visualization of the decomposition algorithm on a single machine scheduling polytope for three jobs

In order to describe the technical details of the algorithm, we use the following notation and facts, all of which follow from the structural insights of Section 5.2.2.

v: vertex of Q corresponding to the permutation $1, 2, \dots, n$; we have $v = v^1$;

J^t: set of indices;

F^t: face of Q_v associated with J^t such that $x_j = x_{j+1}$ for all $x \in F^t$ and all $j \in \{1, \dots, n-1\} \setminus J^t$;

q^t: vertex of F^t;

v^t: vertex of Q such that $q^t = \frac{1}{2}(v + v^t)$;

H^t: point in F^t;

$\tilde{\kappa}_t$: scalar such that $H^t = \tilde{\kappa}_t q^t + (1 - \tilde{\kappa}_t) H^{t+1}$;

κ_t: scalar corresponding to q^t in the convex combination $H = \sum_t \kappa_t q^t$.

λ_t: scalar corresponding to v^t in the convex combination $H = \sum_t \lambda_t v^t$.

Moreover, for ease of notation and without loss of generality, we assume that the given point $H \in Q$ satisfies $H_1 \leq \ldots \leq H_n$.[3]

[3]This comes at the expense of sorting, which costs $O(n \log n)$ time and, thus, falls within the $O(n^2)$ time complexity of the proposed algorithm.

Algorithm 5.2 Decomposition algorithm

```
    Input: processing times $p$, point $H \in Q$ with $H_1 \leq \ldots \leq H_n$
    Output: at most $n$ vertices $v^t$ of $Q$ and coefficients $\kappa_t \in [0,1]$
    $t := 1, H^1 := H, J^1 := \{i \in \{1,\ldots,n-1\} \mid H^1_i < H^1_{i+1}\}$
    let $v$ be the vertex with $v_1 \leq \ldots \leq v_n$
 5: while $J^t \neq \emptyset$ do
        $q^t :=$ VERTEX$(J^t)$
        $v^t := 2q^t - v$
        $\tilde{\kappa}_t := \min_{j \in J^t} (H^t_{j+1} - H^t_j)/(q^t_{j+1} - q^t_j)$
        $H^{t+1} := (H^t - \tilde{\kappa}_t q^t)/(1 - \tilde{\kappa}_t)$
10:     $J^{t+1} := \{i \in J^t \mid H^{t+1}_i < H^{t+1}_{i+1}\}$
        $\kappa_t := (1 - \sum_{\tau=1}^{t-1} \kappa_\tau)\tilde{\kappa}_t$
        $t := t + 1$
    end while
    $q^t := H^t$
15: $v^t := 2q^t - v$
    $\kappa_t := 1 - \sum_{\tau=1}^{t-1} \kappa_\tau$
    $\lambda_1 := \frac{1}{2} + \frac{1}{2}\kappa_1$
    for $\tau \in \{2,\ldots,t\}$ do
        $\lambda_\tau := \frac{1}{2}\kappa_\tau$
20: end for
```

The subroutine VERTEX(J^t) computes the vertex corresponding to the face associated with J^t as follows: Let $J^t(i)$ denote the i-th element in J^t and define $J^t(0) = 1$. Then, for $j \in \{J^t(i), \ldots, J^t(i+1)-1\}$, we compute

$$q^t_j = \sum_{k=1}^{J^t(i)-1} p_k + \frac{1}{2} \sum_{k=J^t(i)}^{J^t(i+1)-1} p_k \quad .$$

Note that vertex q^t can be computed in linear time per iteration by just computing $P^t_i := \sum_{k=J^t(i)}^{J^t(i+1)-1} p_k$ for all i, in time O(n). Then, $q^t_1 = \frac{1}{2}P^t_1$, and for $j \in \{J^t(i), \ldots, J^t(i+1)-1\}$ and $k \in \{J^t(i+1), \ldots, J^t(i+2)-1\}$, the values for q^t are computed iteratively as $q^t_k = q^t_j + \frac{1}{2}(P^t_i + P^t_{i+1})$.

Theorem 5.9. *For any $H \in Q$, Algorithm 5.2 outputs a convex combination of vertices of Q for H in $O(n^2)$ time.*

Proof. Q_v is chosen such that $H \in Q_v$. From line 10 of the algorithm we have that inequalities (5.7) are tight for H^t and all $j \notin J^t$. Thus, if $H^t \in Q_v$ implies that $H^{t+1} \in Q_v$, then $H^t \in f^t$ for all t. By construction, q^t is the vertex of F^t for which (5.7) is tight for all $j \notin J^t$ and (5.8) is tight for all $k \in J^t$. Now suppose $H^t \in Q_v$. Then, of course, H^t and q^t satisfy (5.8), and we have

$$\sum_{j=1}^{k} H^{t+1}_j p_j = \sum_{j=1}^{k} p_j \frac{H^t_j - \tilde{\kappa}_t q^t_j}{1 - \tilde{\kappa}_t} \geq \frac{1}{2}\left(\sum_{j=1}^{k} p_j\right)^2$$

for all $k \in \{1, \ldots, n\}$. Since we have

$$\tilde{\kappa}_t = \min_{j \in J^t} \frac{H_{j+1}^t - H_j^t}{q_{j+1}^t - q_j^t} \quad , \tag{5.10}$$

and both H^t and q^t satisfy inequalities (5.7), we have the following for all $j \in \{1, \ldots, n-1\}$:

$$\begin{aligned} H_{j+1}^{t+1} - H_j^{t+1} &= \frac{H_{j+1}^t - \tilde{\kappa}_t q_{j+1}^t}{1 - \tilde{\kappa}_t} - \frac{H_j^t - \tilde{\kappa}_t q_j^t}{1 - \tilde{\kappa}_t} \\ &= \frac{1}{1 - \tilde{\kappa}_t} \left(H_{j+1}^t - H_j^t - \tilde{\kappa}_t \left(q_{j+1}^t - q_j^t \right) \right) \quad . \end{aligned}$$

It follows from (5.10) that $\tilde{\kappa}_t \geq \frac{H_{j+1}^t - H_j^t}{q_{j+1}^t - q_j^t}$ for all $j \in J^t$. Thus,

$$\begin{aligned} H_{j+1}^{t+1} - H_j^{t+1} &> \frac{1}{1 - \tilde{\kappa}_t} \left(H_{j+1}^t - H_j^t - \frac{H_{j+1}^t - H_j^t}{q_{j+1}^t - q_j^t} \left(q_{j+1}^t - q_j^t \right) \right) \\ &= 0 \quad . \end{aligned}$$

Hence, H^{t+1} satisfies (5.7)–(5.9). From Lemma 5.7, we have $H^{t+1} \in Q_v$ and, therefore, $H^{t+1} \in F^{t+1}$.

In addition, (5.10) ensures that for at least one $j \in J^t$, we have $H_{j+1}^{t+1} = H_j^{t+1}$ and thus $|J^{t+1}| < |J^t|$. Since $|J^1| \leq n-1$, the algorithm terminates after at most $n-1$ iterations. Let t^* be the value of t as the algorithm terminates. Note that $J^{t^*} = \emptyset$ and thus $H^t = c(Q)$, the center of Q. Furthermore, from line 16 of the algorithm, we have $\kappa_{t^*} = 1 - \sum_{j=1}^{t^*-1} \kappa_j$, which implies $\sum_{j=1}^{t^*} \kappa_j = 1$. For $t \in \{1, \ldots, t^* - 1\}$, we have

$$H^t = \tilde{\kappa}_t q^t + (1 - \tilde{\kappa}_t) H^{t+1} \quad .$$

Iteratively applying this equality yields

$$H = \sum_{t=1}^{t^*-1} \prod_{\tau=1}^{t-1} (1 - \tilde{\kappa}_\tau) \tilde{\kappa}_t q^t + \prod_{\tau=1}^{t^*-1} (1 - \tilde{\kappa}_\tau) H^{t^*} \quad .$$

We also have that

$$\begin{aligned} 1 - \sum_{\tau=1}^{t} \kappa_\tau &= 1 - \sum_{\tau=1}^{t-1} \kappa_\tau - \kappa_t \\ &= 1 - \sum_{\tau=1}^{t-1} \kappa_\tau - \tilde{\kappa}_t \left(1 - \sum_{\tau=1}^{t-1} \kappa_\tau \right) \\ &= (1 - \tilde{\kappa}_t) \left(1 - \sum_{\tau=1}^{t-1} \kappa_\tau \right) \quad , \end{aligned}$$

where the second equality follows from line 11 of the algorithm. Applying this equality iteratively yields

$$1-\sum_{\tau=1}^{t}\kappa_\tau=\prod_{\tau=1}^{t}1-\tilde{\kappa}_\tau \ .$$

This gives us the following identity for κ_t:

$$\kappa_t=\tilde{\kappa}_t\left(1-\sum_{\tau=1}^{t-1}\kappa_\tau\right)=\prod_{\tau=1}^{t-1}\left(1-\tilde{\kappa}_\tau\right)\tilde{\kappa}_t \ .$$

So we have

$$H=\sum_{t=1}^{t^*-1}\kappa_t q^t+\left(1-\sum_{\tau=1}^{t^*-1}\right)H^{t^*}=\sum_{t=1}^{t^*-1}\kappa_t q^t+\kappa_{t^*}q^{t^*}=\sum_{t=1}^{t^*}\kappa_t q^t \ .$$

Now since $v^t=2q^t-v$, we have $q^t=\frac{1}{2}(v+v^t)$. From (17) and (19) we have $\lambda_1=\frac{1}{2}+\frac{1}{2}\kappa_1$ and $\lambda_t=\frac{1}{2}\kappa_t$, for $t=2,\dots,t^*$. This yields:

$$\begin{aligned}H&=\sum_{t=1}^{t^*}\kappa_t q^t=\sum_{t=1}^{t^*}\kappa_t\frac{1}{2}(v+v^t)\\&=\sum_{t=1}^{t^*}\kappa_t\frac{1}{2}v+\sum_{t=1}^{t^*}\kappa_t\frac{1}{2}v^t=\frac{1}{2}v+\frac{1}{2}\kappa_1 v+\sum_{t=2}^{t^*}\frac{1}{2}\kappa_t v^t\\&=\lambda_1 v+\sum_{t=2}^{t^*}\lambda_t v^t=\sum_{t=1}^{t^*}\lambda_t v^t \ ,\end{aligned}$$

where the second equality follows from line 7 of the algorithm. From (5.10), we obtain that $\tilde{\kappa}_t$ is non-negative for all $t\in\{1,\dots,t^*\}$. Therefore, also $\kappa_t\geq 0$ and $\sum_{t=1}^{t^*}\lambda_t v^t$ is indeed a correct convex combination.

Since none of the steps within each of at most $n-1$ iterations takes more than $\mathrm{O}(n)$ time, the total computation time of the algorithm is $\mathrm{O}(n^2)$. $\square$

CHAPTER 6

Concluding remarks and open problems

In this thesis we have discussed several results that all share elements of scheduling and game theory. Like always, answering one question can lead to asking another. In this concluding chapter, we briefly put some of the results of this thesis in context and sketch some of the open problems that arise naturally.

In Chapter 2 we show that the price of anarchy for scheduling jobs on related machines, when using SPT as a local scheduling rule and with the total completion time objective, lies between $e/(e-1) \approx 1.58$ and 2. In fact, we prove that the pure price of anarchy is at least $e/(e-1)$, while the robust price of anarchy is at most 2. The latter implies a bound of 2 for the price of anarchy for mixed Nash equilibria, correlated equilibria and course correlated equilibria. We also show that on identical machines, with identical jobs, there is in fact a gap between the pure price of anarchy and the mixed price of anarchy. An interesting question for the related machine scheduling game is what the true values of the price of anarchy for the different equilibria actually are. Related to that, the question where the gap lies, if any, is interesting on its own. It could be between pure Nash and mixed Nash equilibria or somewhere else.

Furthermore, we have seen that SPT as a local scheduling rule is a local coordination mechanism, in the sense that it does not need prior information about all the jobs to determine a schedule locally. We treat some other coordination mechanisms for the related machine scheduling game in the end of Chapter 2. While we have not been able to prove that SPT has a better worst case performances than any other coordination mechanism, we conjecture that this is indeed the case.

In Chapter 3 we show that the mechanism design problem for scheduling jobs, with two dimensional private data, on a single machine can be solved in polynomial time for randomized mechanisms. We did this by using an extended LP formulation, with linear ordering variables, which we showed can be compactified to a polynomial size formulation. We also provide computational results, which establish a gap between IIA and non-IIA, DSIC and BNIC, and between deterministic and randomized implementations. An interesting future path to follow is to worst-case analyze the gaps between the solutions of these different implementations, which we conjecture to be small. Another interesting direction is to analyze if the techniques we used can be applied in other settings. Roughly spoken, this would have to be problems in which the solution can be represented by a linear order. Moreover, it can be conjectured that there are more problems for which a comparable compactification could be applied to yield polynomial size formulations at no loss of generality. Classifying these problems, and thereby generalizing our compactification results would be an interesting research direction.

Chapter 4 is motivated by two simple questions, namely, "Can we describe mechanisms in such a way that they are easy to interpret?" and "Can we design a method that finds such mechanisms quickly, that is, computationally fast?" We think that IIA mechanisms are the right answer. Particularly, because they can be compactly represented by a list of all types of all agents. This representation is polynomial in size, and a polynomial size representation is not known to exist in general. Moreover, it is easy to interpret and to use by participating agents. For computation of an IIA mechanism, based on the list representation, we have seen that simple neighborhoods for local search do a reasonable job, but converge (too) slowly. It may not come as a big surprise that the heuristics we propose, which make use of the full information of the type graph, and corresponding virtual weights perform (much) better. This can be explained from the fact that they utilize more insight into the nature of the problem, specifically the type graph and the shortest paths that yield the minimal payments. However, our understanding of these heuristics is still very limited. For instance, it is an interesting question if there are special cases for which one can obtain results on convergence or performance guarantees. We think that, generally, the design of local search or other practically efficient algorithms for the design of mechanisms is an interesting direction, and our computational results should be seen as a first step in that direction. We think that they clearly show the viability of the approach, which combines two overarching goals. Namely, simplicity of the computed mechanism and practical efficiency. Exploring such ideas for other or more general mechanism design problems seems worthwhile. Another major open question for the deterministic optimal mechanism design problem remains, namely to settle its computational complexity. We believe it is computationally hard, yet a formal proof is lacking.

In Chapter 5 we have proposed two different algorithms that can be used to decompose a point in the scheduling polytope into a convex combination of vertices. The first algorithm finds the intersection of a line with the scheduling polytope in time $\mathrm{O}(n^2 \log n)$. This leads to a $\mathrm{O}(n^3 \log n)$ decomposition algorithm. The second algorithm makes use of the fact that the scheduling polytope is a zonotope and shows that a decomposition can be found even in time $\mathrm{O}(n^2)$. The latter is a linear time algorithm in the straightforward encoding of the output, which is a collection of at most n vectors in $\mathbb{R}^n$. It seems unlikely that an algorithm exists that can do better. The obvious open question is if our algorithm can be generalized for zonotopes in general. In order to do that, we would have to find computationally efficient expressions both for the centers of symmetry and the faces of the resulting barycentric subdivision, induced by these centers. While this is probably not possible in general, it might be worthwhile to specify and analyze classes of zonotopes for which it can be done.

Bibliography

[1] S. Alaei, H. Fu, N. Haghpanah, J. Hartline, and A. Malekian. Bayesian optimal auctions via multi- to single-agent reduction. In *Proc. 13th ACM Conference on Electronic Commerce*, page 17. ACM, 2012.

[2] E. Anshelevich, J. Postl, and T. Wexler. Assignment games with conflicts: Price of total anarchy and convergence results via semi-smoothness. *arXiv preprint arXiv:1304.5149*, 2013.

[3] J. Augustine, N. Chen, E. Elkind, A. Fanelli, N. Gravin, and D. Shiryaev. Dynamics of profit-sharing games. *Internet Mathematics*, 2013.

[4] Y. Azar, K. Jain, and V. Mirrokni. (Almost) optimal coordination mechanisms for unrelated machine scheduling. In *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 323–332. SIAM, 2008.

[5] J. E. Beasley, M. Krishnamoorthy, Y. M. Sharaiha, and D. Abramson. Scheduling aircraft landings – the static case. *Transportation Science*, 34(2):180–197, 2000.

[6] S. Bhattacharya, S. Im, J. Kulkarni, and K. Munagala. Coordination mechanisms from (almost) all scheduling policies. In *Proceedings of the 5th Conference on Innovations in Theoretical Computer Science*, pages 121–134. ACM, 2014.

[7] D. Briskorn and R. Stolletz. Aircraft landing problems with aircraft classes. *Journal of Scheduling*, 17(1):31–45, 2014.

[8] Y. Cai, C. Daskalakis, and S. M. Weinberg. Optimal multi-dimensional mechanism design: Reducing revenue to welfare maximization. In *Proc. 53rd Annual Symposium on Foundations of Computer Science*, pages 130–139. IEEE Computer Society, 2012.

[9] C. Carathéodory. Über den variabilitätsbereich der fourier'schen konstanten von positiven harmonischen funktionen. *Rendiconti del Circolo Matematico di Palermo (1884-1940)*, 32(1):193–217, 1911.

[10] J. C. Carbajal and R. Müller. Implementability under monotonic transformations in differences, 2014.

[11] G. Christodoulou, E. Koutsoupias, and A. Nanavati. Coordination mechanisms. *Theoretical Computer Science*, 410(36):3327 – 3336, 2009. Graphs, Games and

Computation: Dedicated to Professor Burkhard Monien on the Occasion of his 65th Birthday.

[12] R. Cole, J. R. Correa, V. Gkatzelis, V. Mirrokni, and N. Olver. Decentralized utilitarian mechanisms for scheduling games. *Games and Economic Behavior*, 2013.

[13] V. Conitzer and T. Sandholm. Complexity of mechanism design. In *Proc. 18th Conference on Uncertainty in Artificial Intelligence*, pages 103–110. Morgan Kaufmann Publishers Inc., 2002.

[14] R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of scheduling.* Addison-Wesley Publishing Co., 1967.

[15] J. R. Correa and M. Queyranne. Efficiency of equilibria in restricted uniform machine scheduling with total weighted completion time as social cost. *Naval Research Logistics*, 59(5):384–395, 2012.

[16] W. H. Cunningham. Testing membership in matroid polyhedra. *Journal of Combinatorial Theory B*, 36:161–188, 1984.

[17] W. H. Cunningham. On submodular function minimization. *Combinatorica*, 5: 186–192, 1985.

[18] A. Czumaj and B. Vöcking. Tight bounds for worst-case equilibria. *ACM Trans. Algorithms*, 3(1):4:1–4:17, Feb. 2007.

[19] J. Duives, B. Heydenreich, D. Mishra, R. Müller, and M. Uetz. On optimal mechanism design for a sequencing problem. *Journal of Scheduling*, pages 1–15, 2014.

[20] D. Easley and J. Kleinberg. *Networks, crowds, and markets: Reasoning about a highly connected world.* Cambridge University Press, 2010.

[21] W. L. Eastman, S. Even, and I. M. Isaacs. Bounds for the optimal scheduling of n jobs on m processors. *Management Science*, 11(2):pp. 268–279, 1964.

[22] J. Edmonds. Matroids and the greedy algorithm. *Mathematical Programming*, 1(1):127–136, 1971.

[23] P. C. Fishburn. Induced binary probabilities and the linear ordering polytope: a status report. *Mathematical Social Sciences*, 23(1):67 – 80, 1992.

[24] J. Fonlupt and A. Skoda. Strongly polynomial algorithm for the intersection of a line with a polymatroid. In W. Cook, L. Lovász, and J. Vygen, editors, *Research Trends in Combinatorial Optimization*, pages 69–85. Springer, 2009.

[25] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co., 1979.

[26] A. Gershkov, J. K. Goeree, A. Kushnir, B. Moldovanu, and X. Shi. On the equivalence of bayesian and dominant strategy implementation. *Econometrica*, 81(1):197–220, 2013.

[27] R. Graham, E. Lawler, J. Lenstra, and A. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287 – 326, 1979.

[28] M. Grötschel, L. Lovász, and A. Schrijver. The ellipsoid method and its consequences in combinatorial optimization. *Combinatorica*, 1:169–197, 1981.

[29] M. Grötschel, L. Lovász, and A. Schrijver. *Geometric algorithms and combinatorial optimization.* Algorithms and combinatorics. Springer, 1988.

[30] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure, dynamics, and function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, pages 11–15, Aug. 2008.

[31] J. D. Hartline and A. Karlin. Profit maximization in mechanism design, 2007.

[32] B. Heydenreich, R. Müller, and M. Uetz. Games and mechanism design in machine scheduling - an introduction. *Production and Operations Management*, 16(4):437–454, 2007.

[33] B. Heydenreich, D. Mishra, R. Müller, and M. Uetz. Optimal mechanisms for single machine scheduling. In C. Papadimitriou and S. Zhang, editors, *Internet and Network Economics*, volume 5385 of *Lecture Notes in Computer Science*, pages 414–425. Springer, 2008.

[34] R. Hoeksma and M. Uetz. The price of anarchy for minsum related machine scheduling. In R. Solis-Oba and G. Persiano, editors, *Approximation and Online Algorithms*, volume 7164 of *Lecture Notes in Computer Science*, pages 261–273. Springer, 2012.

[35] R. Hoeksma and M. Uetz. Two dimensional optimal mechanism design for a sequencing problem. In M. Goemans and J. Corréa, editors, *Integer Programming and Combinatorial Optimization*, volume 7801 of *Lecture Notes in Computer Science*, pages 242–253. Springer, 2013.

[36] R. Hoeksma and M. Uetz. Optimal mechanism design for a sequencing problem with two dimensional private data. Invited for publication in Operations Research. Under review, 2014.

[37] R. Hoeksma, B. Manthey, and M. Uetz. Decomposition algorithm for the single machine scheduling polytope. In P. Fouilhoux, L. E. N. Gouveia, A. R. Mahjoub, and V. T. Paschos, editors, *Combinatorial Optimization*, volume 8596 of *Lecture Notes in Computer Science*, pages 280–291. Springer, 2014.

[38] R. Hoeksma, B. Manthey, and M. Uetz. Decomposition algorithm for the single machine scheduling polytope. Submitted to Discrete Optimization. Under review, 2014.

[39] R. Hoeksma, H. Nguyen, and M. Uetz. Fast and scalable mechanism design for a single machine sequencing problem with private data. Manuscript, 2014.

[40] E. Horowitz and S. Sahni. Exact and approximate algorithms for scheduling nonidentical processors. *Journal of the ACM*, 23(2):317–327, Apr. 1976.

[41] O. H. Ibarra and C. E. Kim. Heuristic algorithms for scheduling independent tasks on nonidentical processors. *Journal of the ACM*, 24(2):280–289, 1977.

[42] N. Immorlica, L. E. Li, V. S. Mirrokni, and A. S. Schulz. Coordination mechanisms for selfish scheduling. *Theoretical Computer Science*, 410(17):1589 – 1598, 2009. Internet and Network Economics.

[43] S. Iwata, L. Fleischer, and S. Fujishige. A combinatorial strongly polynomial time algorithm for minimizing submodular functions. *Journal of the ACM*, 48 (4):761–777, 2001.

[44] R. M. Karp. Reducibility among combinatorial problems. In R. E. Miller, J. W. Thatcher, and J. D. Bohlinger, editors, *Complexity of Computer Computations*, The IBM Research Symposia Series, pages 85–103. Springer, 1972.

[45] E. Koutsoupias and C. Papadimitriou. Worst-case equilibria. In C. Meinel and S. Tison, editors, *16th Annual Symposium on Theoretical Aspects of Computer Science*, volume 1563 of *Lecture Notes in Computer Science*, pages 404–413. Springer, 1999.

[46] C. W. Lee. Subdivisions and triangulations of polytopes. In *Handbook of Discrete and Computational Geometry*, chapter 17. Chapman & Hall/CRC, 2nd edition, 2004.

[47] A. M. Manelli and D. R. Vincent. Bayesian and dominant-strategy implementation in the independent private-values model. *Econometrica*, 78(6):1905–1938, 2010.

[48] A. Munier, M. Queyranne, and A. S. Schulz. Approximation bounds for a general class of precedence constrained parallel machine scheduling problems. In R. E. Bixby, E. A. Boyd, and R. Z. Ríos-Mercado, editors, *Integer Programming and Combinatorial Optimization*, volume 1412 of *Lecture Notes in Computer Science*, pages 367–382. Springer, 1998.

[49] R. B. Myerson. Optimal auction design. *Mathematics of Operations Research*, 6(1):58–73, 1981.

[50] R. B. Myerson. Utilitarianism, egalitarianism, and the timing effect in social choice problems. *Econometrica: Journal of the Econometric Society*, pages 883–897, 1981.

[51] R. B. Myerson. Game theory: analysis of conflict. *Harvard University Press*, 1991.

[52] J. Nash. Equilibrium points in n-person games. *Proceedings of the National Academy of Sciences of the United States of America*, 36(1):48–49, 1950.

[53] N. Nisan. Introduction to mechanism design (for computer scientists), 2007.

[54] C. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[55] C. Papadimitriou. Algorithms, games, and the internet. In *Proceedings of the Thirty-third Annual ACM Symposium on Theory of Computing*, pages 749–753. ACM, 2001.

[56] C. Papadimitriou and K. Steiglitz. *Combinatorial Optimization: Algorithms and Complexity.* Dover Publications, 1998.

[57] C. Phillips, C. Stein, and J. Wein. Scheduling jobs that arrive over time. In S. G. Akl, F. Dehne, J.-R. Sack, and N. Santoro, editors, *Algorithms and Data Structures*, volume 955 of *Lecture Notes in Computer Science*, pages 86–97. Springer, 1995.

[58] M. Queyranne. Structure of a simple scheduling polyhedron. *Mathematical Programming*, 58(1-3):263–285, 1993.

[59] M. Queyranne and A. S. Schulz. Polyhedral approaches to machine scheduling, 1994.

[60] M. Rahn and G. Schäfer. Bounding the inefficiency of altruism through social contribution games. In Y. Chen and N. Immorlica, editors, *Web and Internet Economics*, volume 8289 of *Lecture Notes in Computer Science*, pages 391–404. Springer, 2013.

[61] O. L. Rivera Letelier. Cotas para el precio de la anarquía de juegos de scheduling. Master's thesis, Universidad de Chile, 2012.

[62] J.-C. Rochet. A necessary and sufficient condition for rationalizability in a quasi-linear context. *Journal of Mathematical Economics*, 16(2):191 – 200, 1987.

[63] T. Roughgarden. Intrinsic robustness of the price of anarchy. In *Proceedings of the Forty-first Annual ACM Symposium on Theory of Computing*, pages 513–522. ACM, 2009.

[64] T. Sandholm. Automated mechanism design: A new application area for search algorithms. In F. Rossi, editor, *Proc. Principles and Practice of Constraint Programming, 9th International Conference*, volume 2833 of *Lecture Notes in Computer Science*, pages 19–36. Springer, 2003.

[65] A. Schrijver. A combinatorial algorithm minimizing submodular functions in strongly polynomial time. *Journal of Combinatorial Theory B*, 80:346–355, 2000.

[66] W. E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3(1-2):59–66, 1956.

[67] A. Turing. On computable numbers, with an application to the entsheidungsproblem. *Proc. London Math. Society*, 2:544–546, 1937.

[68] R. V. Vohra. Optimization and mechanism design. *Mathematical Programming*, 134(1):283–303, 2012.

[69] S. Yasutake, K. Hatano, S. Kijima, E. Takimoto, and M. Takeda. Online linear optimization over permutations. In T. Asano, S.-i. Nakano, Y. Okamoto, and O. Watanabe, editors, *Proc. Algorithms and Computation, 22nd International Symposium*, volume 7074 of *Lecture Notes in Computer Science*, pages 534–543. Springer, 2011.

[70] L. Yu, K. She, H. Gong, and C. Yu. Price of anarchy in parallel processing. *Information Processing Letters*, 110(89):288 – 293, 2010.

[71] G. M. Ziegler. *Lectures on polytopes*, volume 152 of *Graduate Texts in Mathematics*. Springer, 1995.

Samenvatting

Mechanismen voor planningsspellen met egocentrische spelers

Planningsproblemen treden op wanneer meerdere taken moeten worden volbracht. Machineplanningsproblemen zijn wiskundige modellen om dit soort planningsproblemen te beschrijven. Hierbij moet een aantal taken op een aantal machines worden verwerkt. Een oplossing van een machineplanningsprobleem noemt men een schema.

In dit proefschrift worden machineplanningsproblemen behandeld waarbij iedere taak beheerd wordt door een speler. Deze spelers maken individueel keuzes die invloed hebben op het uiteindelijke schema. Hierbij wordt aangenomen dat iedere speler egocentrisch handelt naar zijn eigen doelfunctie. Onder deze aannames worden planningsproblemen met behulp van speltheoretische technieken geanalyseerd. In Hoofdstuk 1 worden de belangrijkste concepten rondom machineplanningsproblemen, speltheorie en optimalisatieproblemen uitgelegd.

Hoofdstuk 2 geeft een analyse van de *price of anarchy* van planningsspellen op basis van een klassiek planningsprobleem met machines die ieder een verschillende snelheid kunnen hebben en taken die ieder een verschillende benodigde hoeveelheid werk kunnen hebben. Hierbij kiezen de spelers op welke machine hun taak zal worden verwerkt. Een planningsregel op de machines bepaalt vervolgens het uiteindelijke schema. De focus ligt op planningsspellen waarbij de machines ieder de *minste werk eerst* planningsregel gebruiken, verder komen ook een aantal andere planningsregels aan bod. Het hoofdresultaat is een bovengrens van 2 voor de *price of anarchy* voor de *minste werk eerst* regel. Dit betekent dat, als spelers zelf hun machine kiezen, het resulterende schema kosten heeft die niet groter zijn dan twee maal de kosten van het optimale schema. Er wordt ook een ondergrens gegeven van $e/(e-1) \approx 1.58$ voor hetzelfde model.

In Hoofdstuk 3 en Hoofdstuk 4 worden planningsproblemen behandeld waarbij sprake is van privé-informatie. In deze setting moeten een aantal taken verwerkt worden op een enkele machine. Hierbij is zowel de benodigde hoeveelheid werk als het gewicht van een taak privé aan de spelers. Deze privé-informatie wordt het type genoemd. Onder de aanname dat spelers gecompenseerd moeten worden voor hun wachttijd, en dat de kosten voor die wachttijd bepaald wordt door het gewicht van de taak, is het doel om een mechanisme te vinden dat de totale gemaakte betalingen aan de spelers minimaliseert. Een mechanisme bestaat hier uit een planningsregel en een betalingsregel, die, op basis van het door de spelers gerapporteerde type, een schema geven van de taken op de machine en een betaling aan iedere speler. Het mechanisme moet er daarbij rekening mee houden dat de spelers, om hun eigen winst te verhogen, niet altijd de waarheid hoeven te spreken over hun type.

Hoofdstuk 3 behandelt de complexiteit van het probleem met privé-gewichten en privé-werkhoeveelheden. Er wordt bewezen dat het vinden van een optimaal mechanisme, waarbij loterijen over verschillende schema's toegestaan zijn, mogelijk is in polynomiale tijd. Dit resultaat wordt behaald met behulp van lineair programmeren (LP). Hierbij wordt een exponentiëel grote LP beschrijving van een relaxatie van het probleem succesvol vertaald naar een polynomiaal grote LP beschrijving, zonder dat daarbij een verlies in prestatie optreedt. De uitkomst is een LP bechrijving die zogenaamde tussenoplossingen bepaalt, in dit geval verwachte starttijden van de taken. De laatste stap is om deze tussenoplossingen te vertalen naar een loterij over deterministische schema's. Hiervoor is een decompositie van een punt in het planningspolytoop naar een convexe combinatie van hoekpunten nodig. Ook dit laatste kan efficiënt gevonden worden.

Het in Hoofdstuk 3 beschreven resultaat beantwoordt de vraag wat de complexiteit is van het bepalen van een optimaal gerandomiseerd mechanisme. Het is echter onduidelijk hoe de procedure gederandomiseerd kan worden. De complexiteit van het bepalen van een optimaal deterministisch mechanisme is een nog open vraagstuk. Het is zelfs niet duidelijk of dit probleem zich in de complexiteitsklasse $\mathcal{NP}$ bevindt. Daarom behandelt Hoofdstuk 4 hetzelfde probleem als Hoofdstuk 3 met een extra voorwaarde, die *onafhankelijkheid van irrelevante alternatieven* wordt genoemd. Deze voorwaarde vereist, in het geval van het enkele machine planningsprobleem, dat de onderlinge volgorde van twee taken alleen afhankelijk is van de types die de spelers behorende bij die taken rapporteren. Met deze extra voorwaarde bevindt het probleem zich in de klasse $\mathcal{NP}$, omdat ieder mechanisme dat aan de voorwaarde voldoet kan worden beschreven als een lijst van de types van de taken. Er wordt aangetoond hoe dit laatste algoritmisch kan worden gebruikt en empirische resultaten met behulp van *local search* en andere constructieve methoden worden gepresenteerd. In de experimenten wordt aangetoond dat deze methodes inderdaad snel zijn en oplossingen vinden die dicht bij een optimaal mechanisme zitten. Dit is zelfs het geval voor zeer grote instanties, die met de LP technieken uit Hoofdstuk 3 zeer moeilijk op te lossen zijn.

In Hoofdstuk 5 wordt de decompositie van een gegeven punt in het planningspolytoop naar een convexe combinatie van hoekpunten behandeld. Dit probleem komt naar voren in de context van het ontwerp van mechanismen, zoals in Hoofdstuk 3, maar het blijkt ook een interessant probleem op zich te zijn. Een combinatorisch tijd $O(n^2 \log n)$ algoritme wordt beschreven dat de doorsnede van een lijn en het planningspolytoop bepaalt. Uit dit algoritme volgt een tijd $O(n^3 \log n)$ algoritme voor het decompositieprobleem. Het hoofdresultaat van dit hoofdstuk laat zien dat een nog sneller algoritme mogelijk is. Namelijk, een algoritme dat in tijd $O(n^2)$ de decompositie vindt. Hierbij wordt gebruik gemaakt van het feit dat het planningspolytoop een zogenaamd *zonotope* is.

Dit proefschrift eindigt in Hoofdstuk 6 met conclusies en een beschrijving van mogelijke vervolgonderzoeken.

About the Author

Ruben Hoeksma was born on March 6, 1985, in Nijmegen, the Netherlands. He received his VWO diploma, with *profiel Natuur & Techniek* and *Natuur & Gezondheid*, in 2003 from R.S.G. Pantarijn in Wageningen. From 2003 to 2010 he studied Applied Mathematics at the University of Twente. He completed a bachelor thesis with the title "Het Dynamische Speelsterkte Systeem" and a master thesis with the title "Price of Anarchy for Machine Scheduling Games with Sum of Completion Times Objective". During his study he obtained a minor in Computer Science and did an internship at Reggefiber, where he worked on optimizing the size of Area-POPs for fiber-optic networks.

In October 2010 Ruben Hoeksma started his Ph.D. research under supervision of Prof.dr. M.J. Uetz on the topic of optimal mechanism design for scheduling problems. During his Ph.D. period he was part of the organizational committee of both the Dutch Mathematical Congress (NMC) 2011 and the 12th Cologne Twente Workshop (CTW 2013). Next to published papers in proceedings of WAOA 2011, IPCO 2013 and ISCO 2014, that are the basis for this thesis, he has a published graph theoretical paper in the proceeding of WAOA 2013 on Approximability of Connected Factors.

CTIT Ph.D. Thesis Series No. 14-342
ISSN: 1381-3617
ISBN: 978-90-365-3827-5