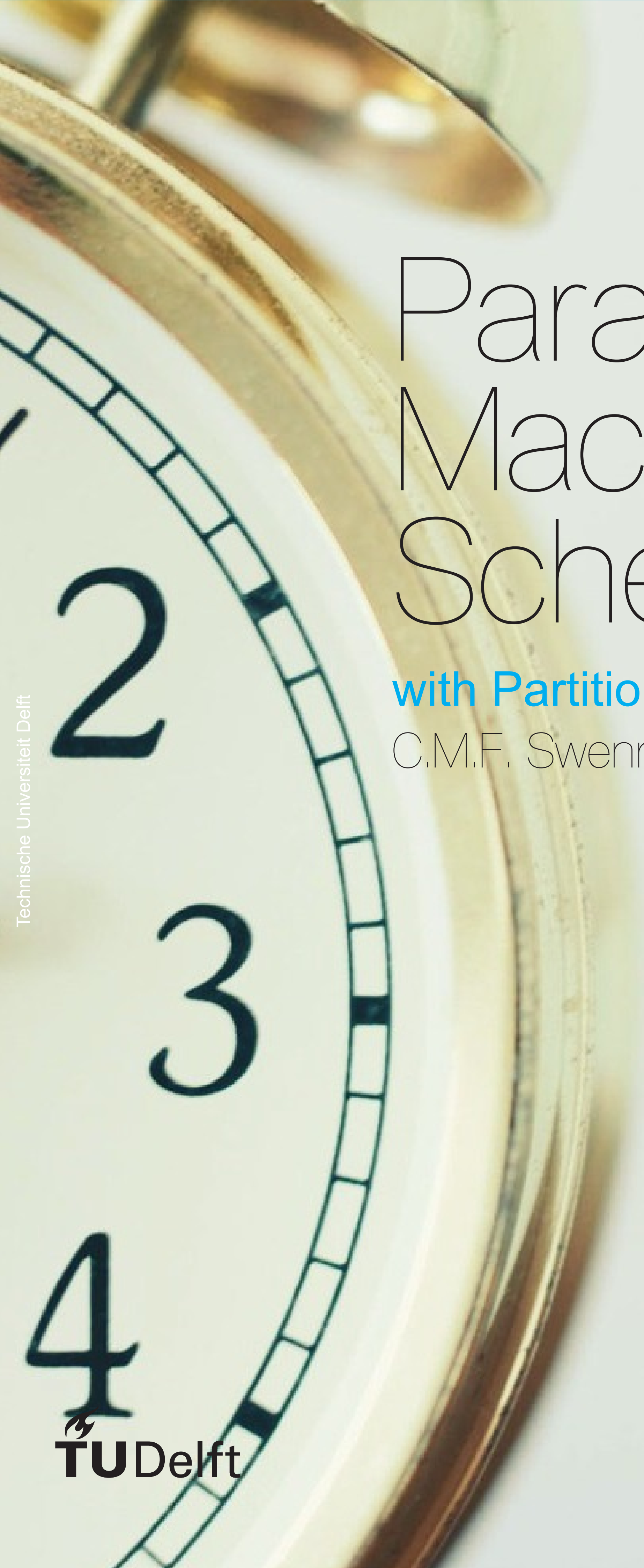


Parallel Machine Scheduling

with Partition Constraints

C.M.F. Swennenhuis

Technische Universiteit Delft

TU Delft

Parallel Machine Scheduling

with Partition Constraints

by

C.M.F. Swennenhuis

to obtain the degree of Master of Science Applied Mathematics
at the Delft University of Technology,
to be defended publicly on Thursday October 18, 2018 at 10:00 AM.

Student number:	4149645	
Project duration:	January 18, 2018 – October 18, 2018	
Thesis committee:	Prof. dr. ir. K.I. Aardal,	TU Delft, Responsible Full Professor
	Dr. ir. L.J.J. van Iersel,	TU Delft, supervisor
	Ir. T.M.L. Janssen,	TU Delft, supervisor
	Dr. C. Kraaikamp,	TU Delft

Acknowledgements

This thesis project is the final part of my Master Applied Mathematics at Delft University of Technology.

I would like to thank my supervisors ir. T.M.L. Janssen and dr.ir. L.J.J. van Iersel for their excellent supervision. During our many meetings, I was able to ask any questions and always felt re-inspired to work on my research again. Because of their supervision, I felt comfortable to try many new approaches and felt supported throughout the whole project. I'm also thankful for giving me the opportunity to contribute to the paper.

Furthermore, my gratitude goes to prof.dr.ir. K.I. Aardal for listening to my wishes for my graduation project. As with my bachelor thesis, she assigned me to an extremely interesting project with great supervisors.

Next, I would like to thank dr. C. Kraaikamp for being part of my thesis committee and for the years of being a great teacher and help throughout the years of my studies.

Finally, I am grateful for the help of many others who helped during my research, such as Dr. D.C. Gijswijt for his remarks on the paper, T. Bosman for his many definitions and insights on the problem and my friends and family for always supporting me.

C.M.F. Swennenhuis
Delft, October 2018

Contents

List of Figures **1**

1 Introduction **3**
1.1 Background . . . 3
1.2 Notation . . . 3
1.3 Previous Work . . . 4
1.4 Structure of the Thesis . . . 5

2 Theoretical Results **7**
2.1 Introduction . . . 7
2.2 Definitions . . . 9
2.3 Problem Properties . . . 10
2.4 Shortest Processing Time First . . . 15
2.5 Machine Subset Constraints . . . 19
2.6 Unmovable Resources . . . 22
2.7 Two Resources per Job . . . 23
2.8 Conclusion . . . 24

3 Exact Algorithms **25**
3.1 ILP with Starting Time Variables . . . 25
3.2 ILP with Preemptions . . . 26
3.3 ILP with Precedence Constraints . . . 27
3.4 MILP with Precedence Constraints . . . 28
3.5 Dynamic Programming . . . 29

4 Heuristics **31**
4.1 SPT-*available* . . . 31
4.2 Semi-SPT-*available* . . . 31
4.3 Dynamic Programming Heuristics . . . 31
4.4 LP Relaxations . . . 32

5 Experimental Results **33**
5.1 Implementation . . . 33
5.2 Test Data . . . 34
5.3 Exact Algorithms Results . . . 34
5.4 Heuristics Results . . . 37
5.5 Conclusion . . . 41

6 Conclusion and Future Work **43**
6.1 Conclusion . . . 43
6.2 Future Work . . . 44

Bibliography **45**

List of Figures

2.1 Proof of Theorem 11, situation where $i = i'$. 12
2.2 Proof of Theorem 11, finding a better solution if $C_{j'} \leq t_2$ 12
2.3 Proof of Theorem 11, only possible situation in an optimal schedule with preemption. . . 12
2.4 Proof of Theorem 11, finding a better solution if $|S(j)| < |S(j')|$ 13
2.5 Proof of Theorem 11, finding a better solution if $|S(j)| > |S(j')|$ 13
2.6 Min Cost-Flow instance for $P|$*partition*$, p_j = 1| \sum_j C_j$ 14
2.7 Optimal and SPT-*available* schedule for Example 1 16
2.8 Feasible schedule for reduction in the case of a yes-instance. 20
2.9 Example of the reduction from edge coloring to $P|$*partition*$(2), p_j = 1| \sum_j C_j$ 24

4.1 Example of rescheduling optimal solution in order of completion time does not give optimal solution. 32

5.1 Average running time for $m = 2$, $K = 3$ and $p_{max} = 5$ for the exact algorithms. 35
5.2 Average running time of tested exact algorithms, set 2. 36
5.3 Average running time of tested exact algorithms, set 3 36
5.4 Experimental results for all heuristics $m = 2$, $K = 3$, $p_{max} = 100$. 38
5.5 Average running time of best tested heuristics. 39
5.6 Average accuracy ratio of best tested heuristics. 40

1 Introduction

1.1. Background

In the semiconductor industry, fabrication plants produce wafers containing chips. These chips are used in goods such as smartphones and notebooks. The wafers are built layer by layer, each time going through a production cycle. During one of the stages of the cycle, the wafers go through the photolithography machines. In these machines, a certain geometrical pattern is put onto the wafers with the help of reticles and UV light. Each set of wafers needs a specific reticle for this process and the reticles are unique. It is therefore important to make sure the necessary reticle is available when processing a set of wafers on the machines. The photolithography machines are expensive and often are the main bottleneck in the fabrication plants. Hence, optimizing the use of these machines can improve the overall performance of the fabrication plant.

It is useful to model the optimization of the machines as a scheduling problem. One can view the wafers that need to be processed as a set of jobs, each having its own processing time. The goal is to find a schedule, such that each machine processes at most one job at the same time and a certain objective value is optimized. Since the scheduling of the lithography machines is a continuous process, we choose to minimize the total completion time. This is the sum of the completion times of the jobs. However, since a reticle is needed in the process, we need to make sure that the required reticle is available if a job is being processed. Hence, auxiliary resource constraints are necessary for our scheduling problem, where the reticles may be viewed as a resource.

Note that the processing times in practice may depend on the machine the job is processed on. Furthermore, each job may have a priority value, also called weight. These weights should be taken into account during the scheduling. One would then minimize the total *weighted* completion time. However, in this thesis, we will assume that all processing times rely only on the jobs and all priority values are equal, unless mentioned otherwise. Although a lot of research has been done on all kinds of scheduling problems with auxiliary resource constraints, none of the known problems fully capture our problem, which is therefore an interesting topic of study.

1.2. Notation

Scheduling problems can be classified using the notation from Graham et al. [17]. The notation will be explained briefly and only input relevant for this thesis will be introduced. Assume that J is the set of jobs to be scheduled and $|J| = n$. Suppose that there are m available machines. Then a scheduling problem can be characterized by a 3-field problem classification $\alpha|\beta|\gamma$.

The α field describes the machine environment. Let p_{ij} be the processing time of the job j on machine i. In all the described environments every job only needs to be processed once. Possible inputs are:

- **1**: there is only one machine.
- **P**: parallel machines. This indicates that the machines are identical and thus the processing times only depend on the jobs, i.e. $p_{ij} = p_j$ for each machine i.
- **Pm**: there are m parallel machines available.
- **Q**: related machines. This implies that each machine has a *speed factor* q_i and $p_{ij} = q_i p_j$.
- **R**: unrelated machines. There are no restrictions on p_{ij}.

The β field indicates a number of job characteristics. A combination of those can be part of the β field. The following possible inputs will be used:

- *prmp*: Preemptions are allowed. Preempting a job is interrupting a job at a machine to resume it at a later time (and possibly at a different machine). No processing is lost when a job is interrupted.
- *res*: There is a set $R = \{r^1, ..., r^K\}$ of resources and each job j requires the use of r_{kj} units of resource r^k during its processing. There resources are limited, so one must make sure that enough resources are available at any time.
- *prec, tree, chains*: Precedence constraints apply. There is a precedence relation $\prec_P$, such that $j \prec_P j'$ if job j' can only start after job j is finished. The precedence relation can be described as an acyclic directed graph. If this graph is in the form of a tree or a collection of chains, we will identify this as such in the β field.
- $p_{ij} = 1$: Each job has the same processing time.
- *partition*: There is a set $R = \{r^1, ..., r^K\}$ of resources and each job j uses exactly one resource. Only one job using that resource can be processed at the same time. A more formal definition will be given in Section 2.2.
- $\mathcal{M}_j$: Each job has set $\mathcal{M}_j$, which is a subset of machines on which it can be processed, also called processing set restrictions.

The γ field defines the optimality criterion. This is the value that one wants to optimize. In our thesis, only $\sum_j C_j$ and its weighted variant $\sum_j w_j C_j$ will be used, where C_j is the completion time of job j. These two criteria are also known as minimizing the *total completion time* and the *total weighted completion time*.

Using this notation, the problem $P|$*partition*$|\sum_j C_j$ will be studied in this thesis.

1.3. Previous Work

It is well-known that $P||\sum_j C_j$ is solvable in $\mathcal{O}(n \log n)$ time by sorting the jobs in order of shortest processing times (SPT) and then scheduling them in that order at the first possibility. The problem with unrelated machines ($R||\sum_j C_j$) is solvable in $\mathcal{O}(n^3)$ time, as it can be formulated as an $m \times n$ transportation problem [9]. However, scheduling problems minimizing the total completion time tend to become hard quite quickly. For example, if we look at minimizing the weighted total completion time, $1||\sum_j w_j C_j$ can be solved by scheduling the jobs in order of shortest weighted processing times (p_j/w_j). This takes $\mathcal{O}(n \log n)$ time, since the jobs must be sorted [26]. However, $2||\sum_j w_j C_j$ is already $\mathcal{NP}$-hard as shown by Lenstra et al. [21]. Hence, also $P||\sum_j w_j C_j$ is $\mathcal{NP}$-hard.

Precedence constraints are also widely studied in scheduling theory. $P|\textit{prec}|\sum_j C_j$ is known to be $\mathcal{NP}$-hard. The result is even stronger, as $P|\textit{chains}|\sum_j C_j$ is also $\mathcal{NP}$-hard, as shown by Du et al. [13]. The reduction for this is from 3-PARTITION. In the proof, the chains are sorted in order of processing time, from largest to smallest. This detail is important for our research, as $P|\textit{partition}|\sum_j C_j$ will be proven to be a special case of precedent constraints in the form of chains. The chains are then structured from smallest to largest processing time.

Sometimes, allowing preemptions does not change the problem. In other words, the optimum of the problem does not change, by allowing jobs to be interrupted. For example McNaughton [24] showed that each instance of $P|\textit{prmp}|\sum w_j C_j$ has an optimal solution without preemptions. Also for for $P|\textit{prmp}, \textit{chains}|\sum w_j C_j$ preemptions are redundant as proved by Du et al. [13]. However, in many cases allowing preemption *does* change the problem. For example, in $Q2|\textit{prmp}, p_j = 1|\sum_j C_j$ it can be more advantageous to schedule jobs partly on a faster machine.

Recall that $R||\sum_j C_j$ is polynomially solvable. Horn [19] showed that allowing processing set restrictions (denoted by $\mathcal{M}_j$), does not make the problem harder. This makes sense, as one can just set the processing times to infinity for p_{ij} if job j cannot be scheduled on machine i. This also implies that $P|\mathcal{M}_j|\sum_j C_j$ is also polynomially solvable. Allowing preemptions, however, reveals something very interesting. It has been shown that for $P|\mathcal{M}_j, \textit{prmp}|\sum_j C_j$ preemptions are redundant [8], whereas $R|\mathcal{M}_j, \textit{prmp}|\sum_j C_j$ becomes $\mathcal{NP}$-hard [25]. This is remarkable, since allowing preemptions does not often make a problem harder.

1.4. Structure of the Thesis

This thesis is split into three main parts. The first part is a paper which was written in collaboration with the supervisors during the period of the master thesis. The paper mainly contains theoretical results about the scheduling problem. These results include fundamental properties of the problem, an analysis of a greedy algorithm and three related problems which are shown to be $\mathcal{NP}$-hard. Although the complexity of the problem on parallel machines remains open, we conjecture that the problem with unrelated machines is also $\mathcal{NP}$-hard. The second part of the thesis discusses several exact algorithms for solving the problem, which are a number of (mixed) integer linear programs. The third part focuses on heuristics, such as greedy algorithms, dynamic programming algorithms and LP relaxations. The efficiency of these exact algorithms as well as the efficiency and the accuracy of the heuristics will then be assessed in the last chapter.

2

Theoretical Results

This chapter is based on a article by Janssen et al. [20]. Parts of this article were written during the period of the master thesis and focus mainly on theoretical results on the problem. My contributions were mainly to the following theorems:

- Theorem 11
- Theorem 12
- Lemma 18
- Theorem 19
- Corollary 20
- Theorem 22
- Corollary 23

Furthermore, in comparison to Janssen et al. [20], Corollary 24 and Theorem 25 were added, as well as some extra text after Theorem 12 explaining the importance of the theorem.

2.1. Introduction

We study a variant of the problem of scheduling jobs on parallel machines with the objective to minimize the total completion time, i.e. the sum of the completion times of all jobs. In this variant, each job uses exactly one resource, thus partitioning the jobs. There is only one unit of each resource available at any time, so jobs using the same resource cannot be processed simultaneously. Using the classification of Graham et al. [17], we will denote the problem as $P|\textit{partition}|\sum_j C_j$. In this classification, scheduling problems are classified by parameters $\alpha|\beta|\gamma$ where the α-field describes the machine environment, the β-field indicates job characteristics and the γ-field reflects the optimality criteria.

The problem is motivated by a scheduling problem found in the semiconductor industry. The wafer, which contains the chips, will visit different production bays multiple times during its production cycle. The expensive photolithography pieces of equipment are often the bottleneck of the production line. Hence, the overall performance of the factory can be improved by raising the equipment throughput on these tools (which is achieved by minimizing $\sum_j C_j$). Photolithography is a process to transfer the geometric pattern of a chip-design onto a wafer. This is done by putting light through a reticle onto the production wafer. This reticle contains the geometrical pattern of the computer chip. Thus, when trying to schedule jobs in the photolithography bay, we need to make sure the reticle (the resource) is available when a job is processed.

$P||\sum_j C_j$ is the problem of scheduling jobs on parallel machines, where one wants to minimize the total completion time. It is well-known from the literature that $P||\sum_j C_j$ is polynomially solvable by using the shortest processing time first (SPT) order on the earliest available machine (Conway et al. [11]). This rule makes sure that every time a machine finishes a job, it will be assigned, from among the jobs waiting, the job with a shortest processing time.

Our problem adds auxiliary resource constraints. Blazewicz et al. [5] describe the resource requirements with the entry res$\lambda\delta\rho$ in the β field of the scheduling problem. The number of different resources is given by $\lambda \in \{\cdot, c_\lambda\}$. If $\lambda = c_\lambda$, the number of resources is given by c_λ. If $\lambda = \cdot$ it is part of the input. The resource capacities are denoted by $\delta = \{\cdot, c_\delta\}$. If $\delta = c_\delta$, there is exactly c_δ of every resource available. If $\delta = \cdot$, the total amount available of a resource is part of the input. The resource requirements per job are denoted by $\rho = \{\cdot, c_\rho\}$. If $\rho = c_\rho$, every job needs exactly c_ρ of a resource it requires. If $\rho = \cdot$, the amount required is part of the input.

The type res $\cdot 11$ implies that per resource type, there is one resource available at any given time and this resource will be used entirely if a job needing this resource is processed. This implies that jobs that share the same resource cannot be processed simultaneously. When only res $\cdot 11$ is in the β field of the scheduling problem, a job can need any number of resources. This does not capture that every job in the lithography bay needs only one resource, the reticle. Therefore, we indicate the problem within this paper by *partition* in the β field of the scheduling problem. Hence, *partition* is a special case of res $\cdot 11$ resources.
We know from Blazewicz et al. [5] that if the number of machines is $m \geq 3$ and there are no further restrictions on the $P|$res$\cdot 11, p_j = 1|\sum_j C_j$, the problem is $\mathcal{NP}$-hard. The proof is based on a reduction from partition into triangles and uses multiple resources per job. It is proven by Garey and Johnson [14] that $P2|$res$\cdot 11, p_j = 1|\sum_j C_j$ can be solved in polynomial time by a reduction to the matching problem.

The problem can also be viewed as a special case of $PD|$res $1 \cdot 1|\sum_j C_j$. In $PD|$res $1 \cdot 1|\sum_j C_j$, we have dedicated machines and are given only one resource of a certain quantity c_δ and every job needs exactly 1 from that resource. We can rewrite $P|$*partition*$|\sum_j C_j$ to $PD|$res $1 \cdot 1|\sum_j C_j$ by taking c_δ equal to the number of machines and introducing a dedicated machine for every resource, such that all jobs that share a resource have to be processed on the same machine. One could also view $P|$*partition*$|\sum_j C_j$ as a special version of scheduling with conflicts. In scheduling with conflicts, we have again parallel machines, the total completion time objective and jobs cannot be processed at the same time if they share an edge in the conflict graph $G = (J, E)$ with an edge between two jobs if they share the same resource. In our problem, G is a collection of cliques.

Our contribution is as follows. First we prove that allowing preemptions to the problem does not change the problem. Using that, we conclude that in each optimal schedule for $P|$*partition*$|\sum_j C_j$, all jobs sharing the same resource must be processed in order of processing time. Restricting the problem to $p_j = 1$ is proven to be polynomially solvable. Thereafter we look at an approximation algorithm for $P|$*partition*$|\sum_j C_j$, based on a variant of the shortest processing time (SPT) rule that takes the partition constraints into account. We prove that it gives a $(2-\frac{1}{m})$-approximation and show that it cannot give an α-approximation with $\alpha < \frac{4}{3}$. In the last three sections we look at three related problems and show that they are $\mathcal{NP}$-hard. The first problem has additional processing set restrictions for resources, meaning each resource has a set $\mathcal{M}_r$ of machines on which it can be used. From this, we can also conclude that the problem with unrelated machines, i.e. $R|$*partition*$|\sum_j C_j$, is also $\mathcal{NP}$-hard. This is the situation of the scheduling of Photolithography machines in practice. The second related problem assumes that resources are unmovable, meaning that once a resource is used on a machine, it can thereafter only be used on that specific machine. In the last related problem, each job has at most q resources with $q \geq 2$ a constant.

2.2. Definitions

Before we begin our analysis of the problem and its solutions, we first formally define *partition* as part of the β field and some intermediary concepts. We have m identical machines and let J be the set of jobs that are to be scheduled. Each $j \in J$ has a *processing time* p_j and each machine can only process a single job at a time. We will denote C_j as the *completion time* of job j in a feasible schedule for an instance. We want to minimize the sum of the completion times (total completion time).

Definition 1. If *partition* is in the β field, there is a partition R of J, i.e., there is a collection of subsets $R = \{r^1, ..., r^s\}$ with $r^k \subseteq J$, where every job is contained in exactly one of the subsets. If $j, j' \in r^k$, j and j' cannot be processed at the same time. Furthermore, we want to define which resource is used by which job. Let $r_j = \{r^k \in R \mid j \in r^k\}$, i.e., all subsets that contain job j. If two jobs share the same resource, we will denote this by $r_j = r_{j'}$, which implies that $r_j \cap r_{j'} \neq \emptyset$.

When we look at a job, we will often consider the other jobs that share the same resource. We will, therefore, introduce the concept of slack, which, intuitively, is the amount of time before and after the job that its resource is not being used.

Definition 2. A job j has *positive slack* $d^+ \geq 0$, which is the largest non negative number, such that in a given schedule all jobs $j' \in J$ which have the same resource and start after job j. More formally, we define d^+ as

$$d^+ := \min \left\{C_{j'} - p_{j'} - C_j | j' \in J \text{ satisfies } r_{j'} = r_j \text{ and } C_{j'} > C_j\right\}$$

where we define $+\infty$ as the minimum over the empty set.
Similarly, a job j has *negative slack* $d^- \leq 0$, which is the largest non negative number, such that all jobs $j' \in J$ which have the same resource and start before job j, finish at least d^- time units before j starts. More formally,

$$d^- := \min \left\{C_j - p_j - C_{j'} | j' \in J \text{ satisfies } r_{j'} = r_j \text{ and } C_{j'} < C_j\right\}$$

where we define $+\infty$ as the minimum over the empty set.
The *slack* $d > 0$ of a job j, we define as $d = \min\{d^+, d^-\}$.

We defined the slack of a job by considering all jobs that share the same resource, but often we are only interested in the last job before and the first job after a job j that use the same resource. We therefore introduce the following concept.

Definition 3. Let d^+ be the positive slack of j, we call a job pair (j, j') a *blocking pair* if $r_j = r_{j'}$ and $C_j = C_{j'} - p_{j'} - d^+$. Thus, j' is the first job to start after C_j that uses the same resource as job j. A blocking pair (j, j') is *tight* if $d^+ = 0$.

Given a tight pair where the two jobs are not on the same machine, it can be advantageous to construct a schedule were they actually are on the same machine. We will call this operation ‘untangling’. Before we define it properly however we need to first define a job’s suffix.

Definition 4. Let job j be processed on machine i. The suffix of job j, $\mathcal{S}(j)$, are all jobs $j' \in J$ that are also processed on machine i with $C_j' \geq C_j$ (excluding job j).

Definition 5. Given a tight pair (j, j'), let i be the machine on which j is processed and i' be the machine on which j' is processed. *Untangling* (j, j') is the operation that changes the schedule by swapping suffices between the machines i and i', i.e., we move j' and $S(j')$ to machine i and $S(j)$ to machine i'.

Since we work with parallel machines, untangling will not change any of the start or completion times of the jobs. Hence, it will not create any resource conflicts and the objective function remains the same.

2.3. Problem Properties

In this section, we consider the structure of optimal solutions. We show that there is no non-trivial idle time in an optimal solution and that given a resource, the jobs using that resources are scheduled from shortest to longest. We will continue by looking at the complexity of the problem. Whether or not $P|\textit{partition}|\sum_j C_j$ is $\mathcal{NP}$-hard remains an open problem, but we can show that when $p_j = 1$ the problem is polynomially solvable. We also show that the problem with preemptions is equal to the problem without preemptions.

We first note that if $|R| < m$ the problem becomes trivial. In that case, one will put all jobs which use the same resource in shortest processing time order on one machine. We continue by looking at idle times in a solution.

Lemma 6. For every instance of $P|\textit{partition}|\sum_j C_j$ there exists an optimal solution that contains no idle times.

Proof. Suppose that, in an optimal schedule with idle times. We begin by untangling all tight pairs. If an idle time remains, we consider the last idle time, which appears on machine i that starts on time t_1 and ends at time t_2. Since we have untangled all tight pairs, the resource used by the job on machine i starting at time t_2 is not used until time t_2. Hence, we can start the processing of this job earlier. We can then schedule the job either at time t_1 or at the last time before t_2 its resource was used. This would reduce the completion time of this job. Therefore, after untangling, there cannot be any idle times. Since untangling does not change completion times there is an optimal schedule without idle times. □

In the proof, we saw it is easy to turn an arbitrary optimal schedule into a schedule where tight pairs (j, j') are processed on the same machine. We call such a schedule a *tight* schedule.

Definition 7. A *tight schedule* is a schedule without any idle time and in which each tight blocking pair (j, j') is executed at the same machine

Notice that untangling results in jobs using the same resource being processed one after another on the same machine. We will call these job sequences *trains*

Definition 8. A *train sequence* $T(j_1)$ in a schedule is a maximal sequence of consecutively jobs $j_1, j_2, ..., j_c$ on the same machine using the same resource.

Notice that a tight schedule only consists of train sequences $T(j_k)$ with nonzero slack between the train sequences of the same resource, where the j_k are the first jobs to be scheduled when a machine changes resource.

We continue by looking at the order in which jobs that use the same resource are processed. We will prove that this is from shortest to longest processing time. We will prove this by first looking at the problem with preemptions, notated by $P|\textit{partition}, \textit{prmp}|\sum_j C_j$. In a preemptive schedule, the total amount of processing done on the job needs to be equal to its processing time (p_j), but jobs can be interrupted at any time and the processing done is not lost. A job can thus be split into multiple parts, possibly processed on different machines. We begin by defining these more precisely.

Definition 9. A *job part* j^l is the lth maximal part of the job j that is processed without interruption on a single machine with positive length. The superscript l will be omitted when it is of no importance. A pair of job parts (j, j') is called a *blocking pair* if $r_j = r_{j'}$ and j' is the first job part to start after j that uses the same resource as job part j. A blocking pair (j, j') is *tight* if job part j' starts at the time j ends, i.e., $d^+ = 0$.

The definitions for blocking pairs, slack, suffix, train sequences, tight schedules and untangling can easily be extended to the case of job parts.

Lemma 10. In an optimal schedule for $P|partition, prmp|\sum_j C_j$ all jobs sharing the same resource must be processed in SPT-order, i.e., if job j and j' both use resource $r \in R$ and $p_j < p_{j'}$ then $C_j < C_{j'}$. Furthermore, if $C_j < C_{j'}$, all job parts of j will be processed before any job parts of j'.

Proof. Suppose we have an optimal schedule S where there is not the case. Then there is a resource $r \in R$ and two jobs using this resource ($r_j = r_{j'}$), job j and job j', with $C_j < C_{j'}$ and $p_j > p_{j'}$. From S we get an ordering of the jobs using resource r. Let $j_{S(r,p)}$ denote the lth job finishing in S using resource r.
Create a new schedule S' which is identical to S except for all jobs using resource r. We remove from S all job parts using resource r. This will remove $t = \sum_{j \in J | r_j = r} p_j$ units of processing from the schedule. We fill these units of processing again with the jobs using resource r but now we process them in an SPT-order. We processes the first job in the ordering $j_{S'(r,1)}$ in the first $p_{j_{S'(r,1)}}$ units of t. We schedule the second job in the ordering $j_{S'(r,2)}$ in the first $p_{j_{S'(r,2)}}$ units of t after $C_{j_{S'(r,1)}}$ and so on until all jobs using resource r are scheduled. The resource r will be used in the same time as in S by only a single job, hence S' is a feasible schedule. Furthermore, it holds that $C_{j_{S(r,1)}} \leq C_{j_{S(r,p)}} \forall p$, since

$$p_{j_{S'(r,1)}} + p_{j_{S'(r,2)}} + ... + p_{j_{S'(r,p)}} \leq p_{j_{S(r,1)}} + p_{j_{S(r,2)}} + ... + p_{j_{S(r,p)}}. \qquad (2.1)$$

Since $p_j > p_{j'}$, equation (2.1) is satisfied with inequality for the l job and S cannot be optimal. □

Theorem 11. There is an optimal schedule for $P|partition, prmp|\sum_j C_j$ without any preemptions.

Proof. Assume not, then take any optimal tight schedule with a minimal amount of preemptions. Let t_1 be the time of the last occurring preemption, let job j be the job that is being interrupted with resource r on machine i, let j^l be the respective job part. Let t_2 be the time that job part j^{l+1} starts on machine i'. Let j' be the job part on machine i that starts at t_1 and let r' be its resource.
We know that $t_2 > t_1$, otherwise we would not have a tight schedule. Furthermore, following from Lemma 10, resource r cannot be used by another job between t_1 and t_2.
We also know that $i \neq i'$ by the following argument illustrated in Figure 2.1. Assume $i = i'$. Take $\epsilon > 0$ as the minimal negative slack of all train sequences between t_1 and t_2 on machine i. Then, one can move all jobs between t_1 and t_2 on machine i ϵ to the front and then split j^l on $t_1 - \epsilon$ and move the second part from $[t_1 - \epsilon, t_1]$ to $[t_2 - \epsilon, t_2]$. Since there is no preemption after t_1, at least one job finishes earlier in this new situation and no jobs finishing later. Thus, the original schedule was not optimal.

We know that job j' cannot end before or on t_2. As illustrated in Figure 2.2, if it would, one could move job j' $\epsilon > 0$ to the front, where ϵ is the negative slack of job j'. This would split the job part j^l on $t_1 - \epsilon$ and moving the part that was executed during the interval $[t_1 - \epsilon, t_1]$ to the back of j'. This leads to a feasible schedule, since we defined t_2 as the time that job part j^{l+1} starts and resource r is not used between t_1 and t_2. Furthermore, since j is a finishing job part and it finishes ϵ earlier, this will lead to a schedule with better objective value.

As a result, there is only possible situation that can occur with the last preemption: The last preemption is at a different machine than where it later continues and job j' starts at t_1 on machine i and does not finish before or on t_2. The partial schedule is shown in Figure 2.3.

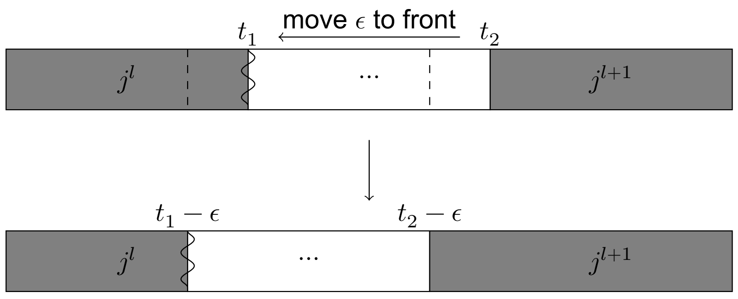


Figure 2.1: Situation where $i = i'$. The squiggly line represents a preemption.

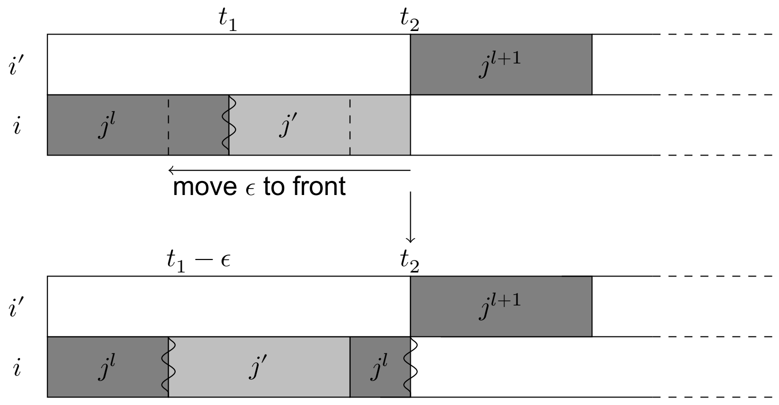


Figure 2.2: Finding a better solution if $C_{j'} \leq t_2$

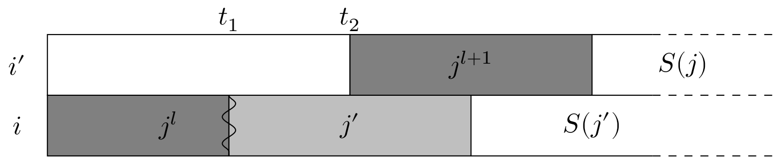


Figure 2.3: Only possible situation in an optimal schedule with preemption.

Let $S(j)$ be job part j^{l+1} on machine i' and its suffix and let $S(j')$ be job part j' on machine i and its suffix. The jobs in these sets all finish, as j^l is the last preemption. When we look at the number of jobs in both sets, there are two possibilities:

- $|S(j)| < |S(j')|$. Figure 2.4 illustrates this case. Take a maximal $\epsilon > 0$ such that all train sequences in $S(j)$ can start ϵ later (i.e. have $d^+ \geq \epsilon$) and all train sequences in $S(j')$ can start ϵ earlier (i.e. have $d^- \geq \epsilon$), while staying in a feasible schedule. Move the sets in the mentioned directions and move the interval $[t_1 - \epsilon, t_1]$ of job j on machine i to machine i' on the interval $[t_2, t_2 + \epsilon]$. Also, move j^{l+1} ϵ to the back and j' ϵ to the front. Clearly, this is a feasible schedule. All job parts in the sets $S(j)$ and $S(j')$ are no preemptions, thus the objective value changes by $\epsilon(|S(j)| - |S(j')|)$, and therefore becomes smaller. Hence, this situation cannot happen in an optimal solution.

- $|S(j)| \geq |S(j')|$. Figure 2.5 illustrates this case. Take a maximal $\epsilon > 0$ such that all train sequences in $S(j)$ can start ϵ earlier (i.e. have $d^- \geq \epsilon$) and all train sequences in $S(j')$ can start ϵ later (i.e. have $d^+ \geq \epsilon$), while staying a feasible schedule. Move the sets in the mentioned directions and move the interval $[t_2, t_2 + \epsilon]$ of job j on machine i' to machine i on the interval $[t_1, t_1 + \epsilon]$. Also move job j^{l+1} to the front and j' to the back. Clearly, this is a feasible schedule. All job parts in the sets $S(j)$ and $S(j')$ are no preemptions, thus the objective value changes by $\epsilon(|S(j')| - |S(j)|)$. Hence, $|S(j)| > |S(j')|$ cannot happen in an optimal solution.

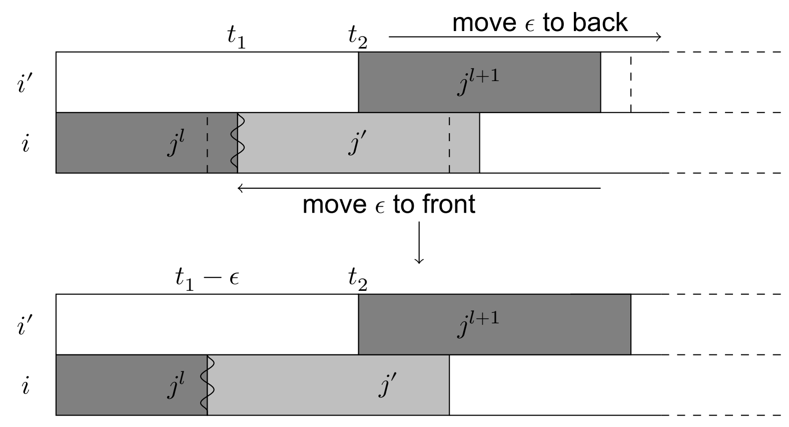


Figure 2.4: Finding a better solution if $|S(j)| < |S(j')|$

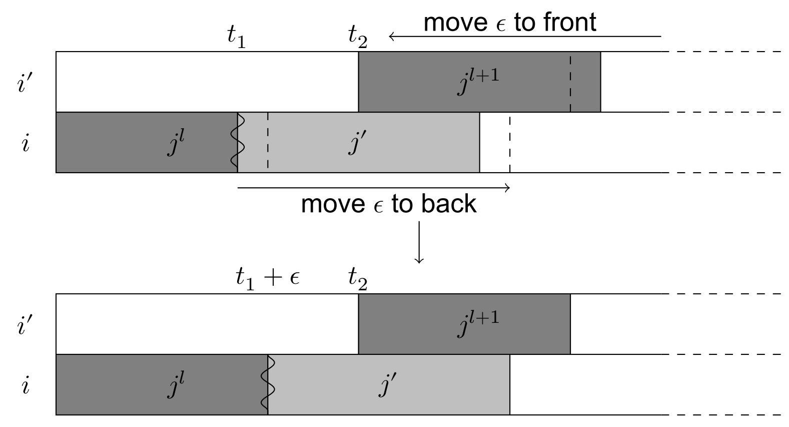


Figure 2.5: Finding a better solution if $|S(j)| > |S(j')|$

Therefore, in an optimal solution, $|S(j)| = |S(j')|$. Since the ϵ was chosen maximal in the $|S(j)| \geq |S(j')|$ case, there must be a new tight pair, created by moving the suffices backward and forward. Untangle new tight pairs, such that the schedule becomes tight again. In the new schedule, it must hold that again $|S(j)| = |S(j')|$, because otherwise the solution was not optimal. It is possible that (one of the) newly tight pairs was (j^l, j^{l+1}), in that case a preemption was removed contradicting the assumption that the number of preemptions was minimal. If not, keep repeating moving the suffices and job parts as described and untangling. This can be done only a finite number of times since there is a maximum number of tight pairs (bounded by n) and because during this process, tight pairs remain tight and no new job parts are created (and with it new preemptions). Hence, there is an optimal schedule containing no preemptions. $\square$

Following from Theorem 11 and Lemma 10, we know that $P|$*partition*$,$*prmp*$|\sum_j C_j$ and $P|$*partition*$|\sum_j C_j$ have the same objective value and we obtain the following result.

Theorem 12. In an optimal schedule for $P|$*partition*$|\sum_j C_j$, all jobs sharing the same resource are processed in SPT-order, i.e., if job j and j' both use resource $r \in R$ and $p_j < p_{j'}$ then $C_j < C_{j'}$.

This has as a consequence, that $P|$*partition*$|\sum_j C_j$ is a special case of $P|$*prec*$|\sum_j C_j$. Without loss of generality we can assume that the jobs are ordered according to their processing times, $j_1 \prec j_2 \prec \ldots \prec j_n$, breaking ties arbitrarily. Then define $j \prec_P j'$ in the precedence graph if and only if $j \prec j'$ and j and j' have the same resource. The problem is even a special case of $P|$*chains*$|\sum_j C_j$, where the precedence graph is a union of chains. We know from literature that $P|\textit{chains}|\sum_j C_j$ is strongly-$\mathcal{NP}$-hard [13]. However, the chains in our precedence graph are in order of processing time, making it impossible to make any conclusions about the complexity of our problem.

We continue by looking at $p_j = 1$, since $P3|\text{res}.11, p_j = 1|\sum_j C_j$ is $\mathcal{NP}$-hard. Surprisingly, with at most one resource per job, the problem becomes polynomially solvable.

Theorem 13. $P|\textit{partition}, p_j = 1|\sum_j C_j$ is polynomially solvable.

Proof. The problem can be reduced to an instance of the Min-Cost Flow problem. Next to the source node s and the target node t, we construct three sets of nodes V_J, $V_{\text{res,pos}}$, $V'_{\text{res,pos}}$ and $V_{M,\text{pos}}$. The set V_J corresponds to the jobs. It has n nodes; one for every job. The set $V_{\text{res,pos}}$ corresponds to resource needed and the completion time/position of the job on a machine in the schedule. Completion time and position on a machine are in this case equal since $p_j = 1$ and we may assume no idle times from Lemma 6. The set $V_{\text{res,pos}}$ has $n|R|$ nodes. The set $V'_{\text{res,pos}}$ is a duplicate of these nodes. The set $V_{\text{M,pos}}$ corresponds to machine used and the completion time/position of the job on the machine in the schedule. It has mn nodes.
We start by constructing arcs with cost 0 and capacity 1. The first set of arcs is constructed from s to every node in V_J. From every node $v_j \in V_J$, we construct an arc to every node in $V_{\text{res,pos}}$ that corresponds to the resource needed by the job j. We construct from every node $v_{r,p} \in V_{\text{res,pos}}$ an arc to the corresponding node with the same resource and position $v'_{r,p} \in V'_{\text{res,pos}}$. Next, we construct from every node $v'_{r,p} \in V'_{\text{res,pos}}$ an arc to every node in $V_{M,\text{pos}}$ having the same position p. Lastly, we construct arcs with capacity 1 and cost equal to position p from every node $v_{m,p} \in V_{M,\text{pos}}$ to t. We thus have $n(1+n+|R|+|R|m+m)$ arcs. Lastly, we require that we have at least n units of flow from s to t.

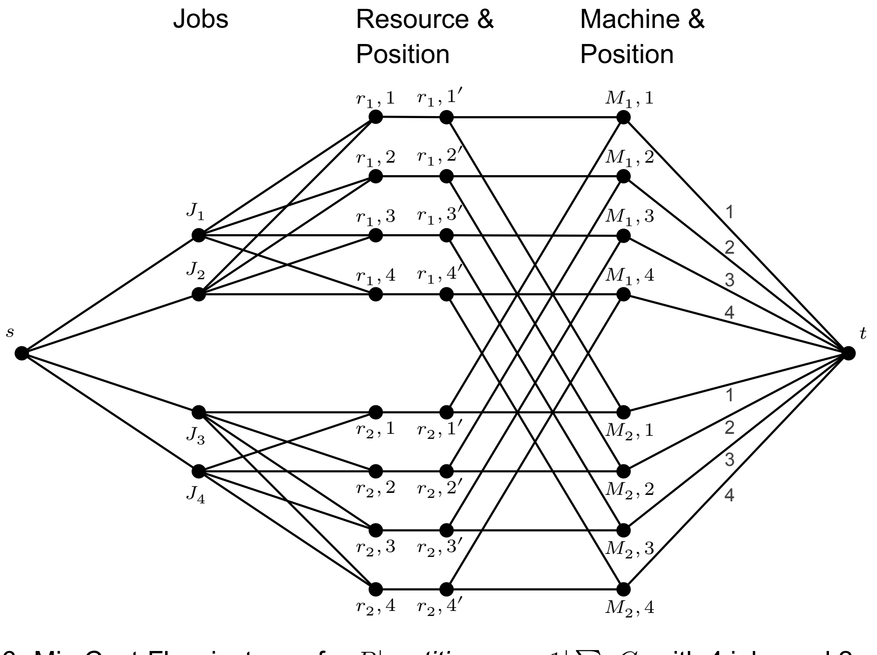


Figure 2.6: Min Cost-Flow instance for $P|\textit{partition}, p_j = 1|\sum_j C_j$ with 4 jobs and 2 resources.

Suppose we have an instance of $P|\textit{partition}, p_j = 1|\sum_j C_j$ with objective value c and consider its corresponding Min-Cost Flow instance. We put one unit of flow in the network for every job in the schedule of $P|\textit{partition}, p_j = 1|\sum_j C_j$ corresponding to the job, its position, which machine and which resource used. For example, for a job j on position p machine M using resource r, we put one unit of flow from s to t through nodes v_J, $v_{r,p}$, $v'_{r,p}$ and M, p. Since no jobs share the same machine or resource at a given position, every node $v \in V \quad \{s,t\}$ will have at most one unit of flow going in and going out. Thus, we have a feasible flow. Furthermore, we only put flow on edges from $v_{m,p} \in V_{M,\text{pos}}$ to t, if and only if we also have a job with corresponding machine m and position p, thus we have a flow of cost c. The reverse is also true by this construction and hence we have an objective value of c for $P|\textit{partition}, p_j = 1|\sum_j C_j$ if and only if we have an objective value of c for its corresponding Min-Cost Flow instance. □

Note that, the above proof can easily be adjusted to the problem where one has more than one unit of a resource available, by setting the capacities of the arcs between $V_{\text{res,pos}}$ and $V'_{\text{res,pos}}$ equal to the amount of resources one has. Furthermore, notice that one could put the costs on a different set of edges. In this way, one can also show that $P|\textit{partition}, p_j = 1|\sum_j w_j C_j$ is polynomially solvable by setting the cost of the edges between v_J and $v_{r,p}$ equal to w_j times the position.

Since we know that $P|\textit{partition}, p_j = 1|\sum_j C_j$ is polynomially solvable, one might wonder what happens when the processing time of each job is bounded, i.e. $1 \leq p_j \leq c$, where c is a constant. A simple Shrinking algorithm would be to create an instance of $P|\textit{partition}, p_j = 1|\sum_j C_j$ by setting all processing times in our original problem to one. We then solve this problem using the construction in Theorem 13 to obtain an optimal schedule $S^{\text{Opt}}_{p_j=1}$. We can then construct a feasible schedule for $P|\textit{partition}, 1 \leq p_j \leq c|\sum_j C_j$ in the following way: All jobs in $S^{\text{Opt}}_{p_j=1}$ start at a given integer $s_j \in \mathbb{Z}^+$, since $p_j = 1$. We can create a feasible solution S_{Alg} from $S^{\text{Opt}}_{p_j=1}$ by setting the starting time for all jobs to cs_j. In this way, in every time interval $ci \leq t \leq c(i+1)$ with $i \in \mathbb{Z}^+$, every machine will only work on one product. There also will not be any resource conflicts, since there were none in $S^{\text{Opt}}_{p_j=1}$ in the corresponding interval $i \leq t \leq (i+1)$.

Proposition 14. The Shrinking algorithm gives a c-approximation for $P|\textit{partition}, 1 \leq p_j \leq c|\sum_j C_j$.

Proof. Let Opt denote the optimal value for $P|\textit{partition}, 1 \leq p_j \leq c|\sum_j C_j$ and S^{Opt} the optimal schedule and $\text{Opt}_{p_j=1}$ denote the optimal value for $P|\textit{partition}, p_j = 1|\sum_j C_j$. The Schrinking algorithm gives a feasible solution of cost $c\text{Opt}_{p_j=1}$, so it suffices to show that $\text{Opt} \geq \text{Opt}_{p_j=1}$. Using S^{Opt}, we can find a schedule $S_{p_j=1}$ for our constructed instance of $P|\textit{partition}, p_j = 1|\sum_j C_j$. This is done by scheduling all jobs 1 time unit before there completion time in S^{Opt}. Thus all completion times in $S_{p_j=1}$ will be the same as in S^{Opt}. Furthermore, since $1 \leq p_j \leq c$ in S^{Opt}, there will be no resource conflict in $S_{p_j=1}$, since there were none in S^{Opt}. □

Note that, one can remove all idle time in S_{Alg} by using the untangling operation found in the proof of Lemma 6.

2.4. Shortest Processing Time First

The shortest processing time first (SPT) rule is optimal for a few scheduling problems. One of which is $P||\sum_j C_j$. In this section, we will look at how well it performs for $P|\textit{partition}|\sum_j C_j$. Before we can do this however we need to adjust the rule slightly to cope with the resources.

Definition 15. The SPT-*available* rule schedules the jobs according to a list. This list contains all jobs ordered for shortest to largest processing time. At any point in time, when a machine is available for processing. The rule selects the first job in the list for which the resource is not in use. It then removes the job from the list. If multiple machines are available at time t and a job is selected of which the resource was not available just before time t, the algorithm will put this job on the machine that was previously using this resource. Otherwise the rule will choose an arbitrary available machine. No job is added to a machine that is available if the resource is in use of all jobs on the list.

Because of the way we defined the SPT-*available* rule, jobs that share the same resource and that are processed one after another will be scheduled on the same machine. In other words, the SPT-*available* rule produces a tight schedule. This schedule also has no idle times, since if at time t a job j of resource r is scheduled on machine m which would create an idle time, then it could not be scheduled on that machine earlier due to some other job j' using the same resource on some machine m'. j can only be scheduled at time t since j' just finished. But the rule states that job j then has a preference for machine m' over m (creating a tight schedule).
The SPT-rule is optimal for $P||\sum_j C_j$. Hence, one might wonder whether this is also the case with the SPT-*available* rule for $P|$*partition*$|\sum_j C_j$. Example 1 shows that this is not the case.

Example 1: (SPT-*available* not optimal)
Consider 2 machines with 12 jobs that use 3 resources. We divide the jobs J into 3 groups depending on the resource used, $J = J_1 \cup J_2 \cup J_3$. We have 4 jobs in $J_1 = \{j_1, \dots, j_4\}$ with $p_1 = \dots = p_4 = 1$ using the first resource. We have another 4 jobs in $J_2 = \{j_5, \dots, j_8\}$ with $p_5 = \dots = p_8 = 1$ using the second resource. Lastly, we have 4 jobs in $J_3 = \{j_9, \dots, j_{12}\}$ with $p_9 = \dots = p_{12} = 1 + \varepsilon$ with $\varepsilon > 0$ using the third resource.

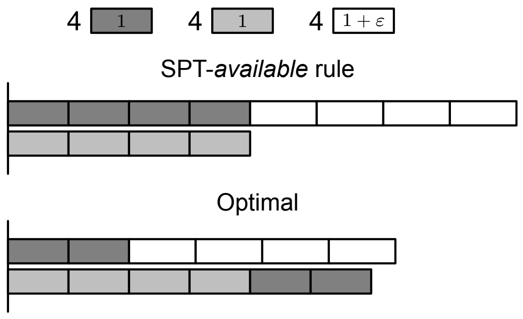


Figure 2.7: Optimal and SPT-*available* schedule for Example 1

The SPT-*available* rule will schedule the jobs in $J_1 \cup J_2$ first on the two machines and will then at time 4 schedule the jobs in J_3 on one machine one after another. This will result in an objective value of $46 + 10\varepsilon$. An optimal schedule would be to schedule first two jobs from J_1 on the first machine and then all jobs from J_3. On the second machine all jobs from J_2 are scheduled first and then the last two jobs from J_1. This results in a schedule with objective value $42 + 10\varepsilon$. Hence SPT-*available* is not optimal.

The SPT-*available* rule is not optimal for $P|$*partition*$|\sum_j C_j$, but it might give a good approximation. Example 1 gives a type of instance that is hard to tackle for the SPT-*available* rule. We generalize this example to find a lower bound on the approximation factor.

Lemma 16. The SPT-*available* rule does not give an α-approximation for $P|$*partition*$|\sum_j C_j$ with $\alpha < \frac{4}{3}$.

Proof. Consider the instance $\mathcal{J}$ with 3 machines and job set $J = J_A \cup J_B$. J_A consists of $3c$ jobs of length 1, that all use their own resource, with $c \in \mathbb{Z}^+$ even. The set J_B consist of $3c$ jobs of length $1 + \varepsilon$ with $\varepsilon > 0$, that all share the same resource, i.e., $r_j = r_{j'}, \forall j, j' \in J_B$.
The SPT-*available* rule will first schedule c jobs from J_A on every machine. Then, it will schedule at time c all jobs from J_B on one machine. Let ALG$_{\mathcal{J}}$ be the objective value of the SPT-*available* rule for instance $\mathcal{J}$. Then,

$$\begin{aligned}\text{ALG}_{\mathcal{J}} &= \tfrac{3}{2}c(c+1) + 3c^2 + \tfrac{3}{2}c(3c+1)(1+\varepsilon) \\ &= 9c^2 + 3c + \tfrac{1}{2}(9c^2 + 3c)\varepsilon\end{aligned}$$

An optimal schedule would schedule the J_b jobs on a single machine and schedule the J_A jobs evenly on the remaining two machine. Let OPT$_{\mathcal{J}}$ be the objective value of the optimal schedule for instance $\mathcal{J}$. Then,

$$\begin{aligned}\text{OPT}_{\mathcal{J}} &= 2\tfrac{3}{4}c(\tfrac{3}{2}c+1)+\tfrac{3}{2}c(3c+1)(1+\varepsilon)\\ &= \tfrac{27}{4}c^2+3c+\tfrac{1}{2}(9c^2+3c)\varepsilon\end{aligned}$$

Thus,

$$\begin{aligned}\frac{\text{ALG}_{\mathcal{J}}}{\text{OPT}_{\mathcal{J}}} &\geq \lim_{c\to\infty}\lim_{\varepsilon\to 0}\frac{9c^2+3c+\frac{1}{2}(9c^2+3c)\varepsilon}{\frac{27}{4}c^2+3c+\frac{1}{2}(9c^2+3c)\varepsilon}\\ &= \lim_{c\to\infty}\frac{9c^2+3c}{\frac{27}{4}c^2+3c}=\frac{4}{3}\end{aligned}$$

□

We will proceed by giving an upper bound to the approximation ratio by using an approach similar to the approach used by Chekuri et al. [10] to show a 2-approximation for the problem of minimizing weighted completion time on m parallel machines with in-tree precedence constraints. We begin by defining the minimum completion time of every job, based on the fact that by Theorem 12 all jobs sharing the same resource must be processed in SPT-order. Without loss of generality we can assume that the jobs are ordered according to their processing times, $j_1 \prec j_2 \prec \cdots \prec j_n$, breaking ties arbitrarily.

Definition 17. The *minimum completion time* k_j for each job j is given by

$$k_j = p_j + \sum_{j' | r_{j'} = r_j \text{ and } p_{j'} \prec p_j} p_{j'}.$$

Define OPT^m as the optimal value for a given instance of jobs on m machines and let OPT$^m_{\text{res}}$ be the optimal value for the instance of jobs on m machines with *partition* constraints. Clearly, OPT^1 $=$ OPT$^1_{\text{res}}$ for each instance since the additional constraints do not interfere with the optimal schedule for one machine. Notice that the optimal schedule for one machine and for parallel machines is created by the SPT rule [11]. Let C^1_j denote the completion time of job j in an optimal schedule using one machine (with or without *partition* constraints) and C^m_j the completion time of job j in an optimal schedule for m machines without *partition* constraints.

Lemma 18. $\frac{1}{m}$OPT^1 $\leq$ OPT^m $\leq$ OPT$^m_{\text{res}}$ for each instance.

Proof. Clearly, the last inequality holds, as an optimal schedule for the problem with constraints is always a feasible solution to the problem without the partition constraints. Hence, we only have to show the first inequality.
Sort the jobs from small to large processing times. Let j be arbitrary but fixed job and let $P_j = \sum_{i=1}^{j} p_i$ be the sum of all processing times of the jobs that have a smaller or equal processing time. Clearly $C^m_j \geq \frac{1}{m}P_j$, since $\frac{1}{m}P_j$ is the earliest time j can finish. We also have that $P_j = C^1_j$, since the SPT-rule is optimal. Thus, for every job j we also have that $\frac{1}{m}C^1_j \leq C^m_j$ from which the first inequality follows. □

We can now prove the upper bound for the SPT-*available* rule.

Theorem 19. The SPT-*available* rule gives a $(2-\frac{1}{m})$-approximation for $P|$*partition*$|\sum_j C_j$.

Proof. Let C^G_j be the completion time of job j in schedule created by the SPT-*available* rule. We begin by proving by induction that

$$C^G_j \leq \left(1-\frac{1}{m}\right)k_j + \frac{1}{m}C^1_j, \quad \forall j \in J. \tag{2.2}$$

The jobs that are the first to be scheduled on a machine are also the first of their resource. Therefore,

$$\begin{aligned} C_j^G &= p_j \\ &= \left(1 - \frac{1}{m}\right) p_j + \frac{1}{m} p_j \\ &= \left(1 - \frac{1}{m}\right) k_j + \frac{1}{m} C_j^1 \end{aligned}$$

in that case.
Now assume that Equation (2.2) holds for any job $j' \prec j$. Then, in particular, it also holds for the job j' that is scheduled right before job j on the same machine. We distinguish the following two cases:

- *j' and j use the same resource..* j is scheduled right after j' and thus

$$\begin{aligned} C_j^G &= C_{j'}^G + p_j \\ &\le \left(1 - \frac{1}{m}\right) k_{j'} + \frac{1}{m} C_{j'}^1 + p_j && \text{(by induction)} \\ &= \left(1 - \frac{1}{m}\right) (k_{j'} + p_j) + \frac{1}{m} (C_{j'}^1 + p_j) \\ &\le \left(1 - \frac{1}{m}\right) k_j + \frac{1}{m} C_j^1 . \end{aligned}$$

- *j' and j use different resources.* This implies that j was scheduled at the first possible free machine and not at a later possibility because of the resource constraints. Define job j'' as the job that is right before j in the schedule for one machine, i.e. the last j'' in the ordering such that $j'' \prec j$. For the starting time of job j (equal to $C_{j'}^G$) it then holds that $C_{j'}^G \le \frac{1}{m} C_{j''}^1$. Using this we see that:

$$\begin{aligned} C_j^G &= C_{j'}^G + p_j \\ &\le \frac{1}{m} C_{j''}^1 + p_j \\ &= \left(1 - \frac{1}{m}\right) p_j + \frac{1}{m} (C_{j''}^1 + p_j) \\ &\le \left(1 - \frac{1}{m}\right) k_j + \frac{1}{m} C_j^1 && \text{(since } p_j \le k_j \, \forall j). \end{aligned}$$

Hence, we can conclude that (2.2) holds for all jobs $j \in J$.

Note that $\sum_j k_j \le OPT_{res}^m$, since $k_j \le C_j$ in any feasible schedule for $P|$*partition*$|\sum_j C_j$ by the definition of the minimal completion time k_j. Using equation (2.2) we get:

$$\begin{aligned} \sum_j C_j^G &\le \sum_j \left(1 - \frac{1}{m}\right) k_j + \sum_j \frac{1}{m} C_j^1 && \text{(using (2.2))} \\ &= \left(1 - \frac{1}{m}\right) \sum_j k_j + \frac{1}{m} \sum_j C_j^1 \\ &\le \left(1 - \frac{1}{m}\right) OPT_{\text{res}}^m + OPT_{\text{res}}^m && \text{(Lemma 18)} \\ &\le \left(2 - \frac{1}{m}\right) OPT_{\text{res}}^m \end{aligned}$$

□

2.5. Machine Subset Constraints

Since we do not know the complexity of $P|partition|\sum_j C_j$, it is interesting to look at related problems. We will look at several of these related problems. We begin by considering the problem where jobs that share the same resource can only be processed on a subset of the machines.

We can add processing set restrictions by adding $\mathcal{M}_j$ to the β field of a scheduling problem, as found in [22]. This means that for each job j, there is a set $\mathcal{M}_j \subseteq \{1, ..., m\}$ such that j can only be scheduled on machines in $\mathcal{M}_j$. Let us define a variation called processing set restrictions for resources as follows: for each resource $r \in \mathcal{R}$ there is a set $\mathcal{M}_r \subseteq \{1, ..., m\}$ such that all jobs sharing resource r can only be scheduled on machines in $\mathcal{M}_r$. We denote these restrictions as $\mathcal{M}_r$ in the β field.

Corollary 20. $P|partition, \mathcal{M}_r, p_j = 1|\sum_j C_j$ is polynomially solvable.

This is a consequence of Theorem 13. One could use the same algorithm, but only include edges $v'_{r,p}$ to $v_{i,p}$ if $i \in \mathcal{M}_r$.

Consider the following NP-complete problem from [15].

Definition 21. 3-PARTITION Given positive integers m and b, and a multiset of $3m$ integers A with $\sum_{a\in A} a = mb$, and $b/4 \leq a \leq b/2$ for all $a \in A$, does there exist a partition $(A_1, ..., A_m)$ of A into 3 element sets such that for each i, $1 \leq i \leq m$, $\sum_{a \in A_i} a = b$?

This problem is NP hard in the strong sense. Using this, we will prove the following theorem.

Theorem 22. $P|partition, \mathcal{M}_r|\sum_j C_j$ is NP hard in the strong sense.

Proof. Assume we have an instance of 3-PARTITION. Since 3-PARTITION is NP-complete in the strong sense, we may assume that mb is bounded by a polynomial in m, which is crucial for our proof. Define $N_c = 2mb$ and $C = 8mb$. Create an instance of $P|partition, \mathcal{M}_r|\sum_j C_j$ with $2m$ machines and the following jobs:

- for all $a \in A$ make job a_j, with processing time $p_{a_j} = a$, a unique resource $r(a_j)$ and $\mathcal{M}_{r(a_j)} = \{1, 2, ..., m\}$. These jobs represent the integers that should be partitioned over the first m machines.
- for all $1 \leq i \leq m$ make N_c jobs called 'C'-jobs with processing time C, resource i and $\mathcal{M}_i = \{i, m+i\}$, so for each i, there are N_c jobs with length C, all sharing the same resource, that can only be scheduled on machines i and $m+i$.
- for all $1 \leq i \leq m$ make job r_i, also called a release date job with processing times $p_{r_i} = b$ and resource i, so it shares its resource with a sequence of 'C'-jobs and can only be scheduled on machines i and $m+i$.
- for all $1 \leq i \leq m$ make job D_i, also called a 'D'-job, with processing time $p_{D_i} = N_C^2 C$, resource $r(D_i)$ and $\mathcal{M}_{r(D_i)} = \{m+i\}$, so its resource is unique and can only be scheduled on machine $m+i$.

Define $Z^+ = mb + m\left(N_C b + (C + N_C C)\frac{N_C}{2}\right) + m(b + N_c^2 C) + 2mb$. We will show that the optimal schedule for the scheduling problem has objective value $Z^* \leq Z^+$ if and only if the 3-PARTITION instance is a yes-instance.

Assume that the 3-PARTITION instance is a yes-instance, then the following schedule is a feasible solution: We can find A_i with $|A_i| = 3$ s.t. $\sum_{a\in A_i}^m a = b$ for all i. Schedule each of these A_i at the beginning of one of the first m machines. Process the jobs from small to big in their processing times per machine. Start the release date jobs r_i at machines $m+i$ at $t = 0$. Process the 'C'-jobs from $t = b$

and onwards at the first m machines. Start each 'D' job D_i at machine $m+i$ at $t=b$. For a visualization of this schedule see Figure 2.8. The objective value of such a solution is equal to

$$Z_{feas} = \underbrace{m \cdot b}_{\text{release date jobs}} + \underbrace{m\left(N_C \cdot b + (C + N_C \cdot C)\frac{N_C}{2}\right)}_{`C' \text{ jobs}} + \underbrace{m\left(b + N_C^2 \cdot C\right)}_{`D' \text{ jobs}} + \underbrace{Z_A}_{a_j \text{ jobs}} ,$$

with $\frac{7}{4}mb \leq Z_A \leq 2mb$ since the following: $\frac{b}{4} \leq a_j \leq \frac{b}{2}$, and the a_j jobs are sorted from small to big in their processing times per machine, so the worst case scenario is if A_i only has jobs of processing times $\frac{b}{3}$ and the best case scenario is if A_i has jobs of processing times $\frac{b}{4}, \frac{b}{4}$ and $\frac{b}{2}$. Since $Z_{feas} \leq Z^+$, we can conclude that $Z^* \leq Z^+$.

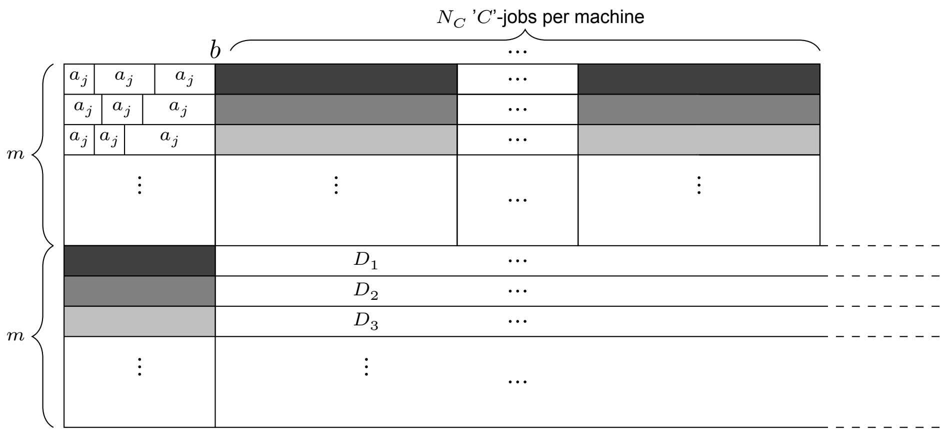


Figure 2.8: Feasible schedule for reduction in the case of a yes-instance.

We will show that if the 3-PARTITION instance is a no-instance, the optimal schedule has an objective value $Z^* > Z^+$. Let $Z^* = Z_r^* + Z_C^* + Z_D^* + Z_a^*$ with Z_r^*,Z_C^*,Z_D^* and Z_a^* the sum of the completion times of the release date jobs, 'C'-jobs, 'D'-jobs and a_j-jobs respectively. Notice that mb is a lower bound on Z_r^* since the jobs cannot start before $t=0$. In the same way, mN_C^2C is a lower bound on Z_D^*. A lower bound on Z_a^* is $\frac{7}{4}mb$, this is because if only the a_j-jobs were to be scheduled on m machines, SPT-order would be optimal and would have 3 jobs on every machine. Suppose not, then there is a machine i_1 with 4 jobs or more. Then there is another machine i_2 with 2 or less jobs. Moving the first job j on machine i_1 to machine i_2 would result in at least 3 jobs finishing p_j earlier and at most 2 jobs finishing p_j later at machine i_2. This leads to a contradiction that SPT is optimal. So we may assume each machine has exactly 3 jobs for finding the lower bound of Z_a^*. Then $\sum_{i=1}^{m}(3a_{i1} + 2a_{i2} + a_{i3})$ is minimal if one chooses all a_{i1} and a_{i2} to be $\frac{b}{4}$, i.e. as small as possible. This implies $a_{i3} = \frac{b}{2}$, as $\sum_{i=1}^{3m} a_j = mb$ and $\frac{b}{4} \leq a_j \leq \frac{b}{2}$ for all j. This leads to a total completion time and therefore a lower bound of $m\left(\frac{b}{4} + \frac{b}{2} + b\right) = \frac{7}{4}mb$ for Z_a^*.

If we have a no-instance, we argue that in the optimal schedule $\exists a_j$ with completion time bigger than b. Assume not, then all first m machines are processing a_j jobs until b, since $\sum_{j=1}^{3m} a_j = mb$. If there is a machine processing more than three a_j jobs, it would have to be four a_j jobs of length $\frac{b}{4}$, so that the a_j jobs are all finished before or at b. But then, there is also a machine processing only two a_j jobs of length $\frac{b}{2}$, otherwise another machine would have to finish its a_j jobs after b. Switching a $\frac{b}{2}$ job with two $\frac{b}{4}$ jobs would then result in a smaller objective value. Hence, all machines are processing exactly three a_j jobs with $\sum_{a_j \in A_i} a_j = b$ for all $1 \leq i \leq m$. Then we would find a partition, leading to a contradiction. So there must be an a_j finishing after b. Since $a_j \in \mathbb{N}$ for all j, we can find an a_j with completion time $\geq b+1$. We distinguish 4 cases:

- *At least one 'C'-job is scheduled before the r_i job with the same resource.* Let N_i be the number of 'C'-jobs on machines i and $m+i$ scheduled before the corresponding release job r_i.

Then r_i starts at N_iC or later. The lower bound on Z_r^* becomes $mb + C \cdot \sum_{i=1}^m N_i$. Then $\sum_{i=1}^m \left((N_C - N_i)b + (C + N_CC)\frac{N_C}{2}\right)$ is a lower bound of Z_C^*. Using the other lower bounds for Z_a^* and Z_D^*, we obtain the following lower bound for Z^*:

$$mb + C \cdot \sum_{i=1}^m N_i + \sum_{i=1}^m \left((N_C - N_i)b + (C + N_CC)\frac{N_C}{2}\right) + mN_C^2C + \frac{7}{4}mb.$$

Then

$$Z^* - Z^+ \geq (C - b)\sum_{i=1}^m N_i - mb - \frac{1}{4}mb > 0,$$

using that $C = 8mb$ and $\sum_{i=1}^m N_i > 0$.

- *All 'C'-job are scheduled after the r_i job with the same resource, but at least one 'C'-job is scheduled on one of the last m machines.* We split this up into two subcases:
  - At least one such 'C'-job is scheduled *before* a 'D'-job on the same machine. We know all 'C'-jobs start after b, hence $Z_C^* \geq m(N_Cb + (N_CC + C)\frac{N_C}{2})$. Then $Z_D^* \geq mN_C^2C + b + C$ since one 'D'-job starts after $b + C$. Using the other lower bounds we get

    $$Z^* - Z^+ \geq b + C - mb - \frac{1}{4}mb > 0,$$

    using that $C = 8mb$.
  - At least one such 'C'-job is scheduled *after* a 'D'-job on the same machine. Let i be the resource of one such 'C'-job and D_i the corresponding 'D'-job. Then the 'C'-job finishes at $N_C^2C + C$ or later, while any 'C'-job scheduled *not* after a D'-job would have a maximum completion time of $\sum_{j=1}^{3m} a_j + b + N_CC$, which is the sum of all processing times of jobs that could possibly be scheduled on machine i. Hence $Z_C^* \geq m(N_Cb + (N_CC + C)\frac{N_C}{2}) + (N_c^2C + C - (mb + b + N_CC))$. Using the lower bounds for Z_r^*, Z_a^* and Z_D^*, we get

    $$Z^* - Z^+ = N_C^2C + C - (mb + b + N_CC) - mb - \frac{1}{4}mb > 0,$$

    using that $C = 8mb$ and $N_C = 2mb$.

- *All 'C'-job are scheduled after the r_i job with the same resource, all 'C'-jobs are scheduled on the first m machines, but least one 'C'-job is scheduled before an a_j job on the same machine.* All 'C'-jobs are scheduled after b, hence $Z_C^* \geq m(N_Cb + (C + N_CC)\frac{N_C}{2})$. At least one a_j has completion time bigger than $C + b$, while for any a_j *not* scheduled after an 'C'-job has a completion time smaller or equal to $\sum_{j=1}^{3m} a_j + b = mb + b$, so $Z_a^* \geq \frac{7}{4}mb + (C + b - mb - b)$. Using the lower bounds for Z_r^* and Z_D^*, we get

  $$Z^* - Z^+ \geq (C - mb) - mb - \frac{1}{4}mb > 0$$

  using that $C = 8mb$.

- *All 'C'-job are scheduled after the r_i job with the same resource, all 'C'-jobs are scheduled on the first m machines and all 'C'-job are scheduled after the a_j jobs on the same machine.* Notice that the feasible solution in Figure 2.8 is structured in a similar way. At least one machine i should have an a_j job with completion time at least $b + 1$. So the sequence of 'C'-jobs on machine i should start at $b + 1$ or later. This means that $Z_C^* \geq m\left(N_Cb + (C + N_CC)\frac{N_C}{2}\right) + N_C$. Using the lower bounds on Z_r^*, Z_a^* and Z_D^*, we get:

  $$Z^* - Z^+ \geq N_c - mb - \frac{1}{4}mb > 0,$$

  using that $N_C = 2mb$.

So if the 3-PARTITION instance is a no-instance, $Z^* > Z^+$, hence the reduction is complete. □

We can use Theorem 22 to show that our original problem with unrelated machines instead of parallel machines is $\mathcal{NP}$-hard. This problem is actually the problem found in the lithography bays of the European semiconductor factories [4].

Corollary 23. $R|$*partition*$|\sum_j C_j$ is $\mathcal{NP}$-hard in the strong sense.

Proof. We can reduce any decision variant instance I_P of $P|partition, \mathcal{M}_r|\sum_j C_j$, asking whether there exists a feasible solution with total completion time smaller than T, to a decision variant instance I_R of $R|partition|\sum_j C_j$ asking the same question. This is done by simply removing the processing set restrictions for resources and changing the processing times to:

$$p_{ij} = \begin{cases} p_j & \text{if } i \in \mathcal{M}_{r(j)} \\ T & \text{if } i \notin \mathcal{M}_{r(j)} \end{cases}$$

where $\mathcal{M}_{r(j)}$ denotes the machine restriction for $r(j)$, the resource of job j. Clearly, any feasible schedule for I_P is also a feasible schedule for the mapped instance I_R with the same total completion time. Hence if we have a yes-instance for I_P, we also have a yes-instance for I_R. However, if we have a no instance for I_P, all feasible solution for I_P have a total completion time at least T. This means that all schedules for I_R processing only j on $i \in \mathcal{M}_{r(j)}$ for all j, also have total completion time at least T. However, any schedule processing at least one j on an $i \notin \mathcal{M}_{r(j)}$ also has a total completion time at least T, because of such a job j. Hence I_R is also a no-instance. □

Theorem 22 also implies the next corollary.

Corollary 24. $P|$*partition*$, \mathcal{M}_j|\sum_j C_j$ is $\mathcal{NP}$-hard in the strong sense.

This can be seen fairly easy, since $P|$*partition*$, \mathcal{M}_r|\sum_j C_j$ is a special case of $P|$*partition*$, \mathcal{M}_j|\sum_j C_j$.

However, if we restrict to two parallel machines and $p_j = 1$ for all jobs, the problem becomes polynomially solvable.

Theorem 25. $P2|$*partition*$, \mathcal{M}_j, p_j = 1|\sum_j C_j$ can be solved in $O(n^3)$.

Proof. This problem will be converted into a maximum matching problem, which can be solved in $O(n^3)$. Assume that $M_j \neq \emptyset$, otherwise no feasible solution exists. Let $j = 1, ..., n$ be the jobs of the instance. Construct an undirected graph $G = (V, E)$ with $V = \{1, ..., n\}$ and E the set of edges for which it holds that $(i, j) \in E$ if jobs i and j can be scheduled at the same time, i.e. i and j use a different resource and $\mathcal{M}_i \cup \mathcal{M}_j = \{1, 2\}$. Find the maximal matching in G. The optimal schedule is then found by first scheduling the found matches. After that, the other jobs are scheduled alone on any possible machine in any order. This is optimal, since changing the order of matches or non-matched jobs does not change the objective value. In no case it's better to schedule a non-matched job before a matched pair. Since the matching we found is maximal, the jobs cannot be rearranged, such that more jobs can be executed simultaneously. □

2.6. Unmovable Resources

Moving the resources can be a costly operation. Thus one might also consider the case were the resources are also fixed on a machine. We therefore consider the problem where every resource can only be used on one machine. We define *unmovable* as an addition to the *partition* constraint, were all jobs $j \in r^k$ have to be processed on the same machine.

Theorem 26. $P|$*partition*$,$ *unmovable*$, p_j = 1|\sum_j C_j$ is $\mathcal{NP}$-hard.

Proof. We give a polynomial time reduction from the 3-Partition problem as defined in Definition 21. We introduce in $P|\textit{partition}, \textit{unmovable}, p_j = 1|\sum_j C_j$ m machines and a number of jobs equal to $n = \sum_{a \in A} a$ and a number of resources equal to $3m$, where M is a large number. For every element of $a \in A$, we associate a number of jobs equal to a sharing the same resource. If we have a yes-instance of 3-Partition, then, $\forall i \in \{1, \ldots, m\}$, we can schedule all jobs associated with $a \in A_i$ to machine i. All machines will then be busy processing until time b. This will give us an objective value $\frac{m}{2}b(b+1)$. If we have a no-instance of 3-Partition, we cannot distribute the jobs evenly over the machines and thus the objective value will be greater than $\frac{m}{2}b(b+1)$. Hence, there exists a solution to 3-Partition if and only if $P|\textit{partition}, \textit{unmovable}, p_j = 1|\sum_j C_j$ as constructed above has an objective value of $\frac{1}{2}mb(b+1)$.
Note that, one might have a yes-instance of $P|\textit{partition}, \textit{unmovable}, p_j = 1|\sum_j C_j$, where on one machine there are 4 resources being used. If this is the case, these resources all have $b/4$ associated jobs and since it is a yes-instance, there also must be a machine using only 2 resources with $b/2$ associated jobs. An easy switch of the last $b/2$ units of processing of these two machine, will turn this also in a yes-instance for 3-Partition. □

2.7. Two Resources per Job

Because the problem was motivated by the scheduling problem found in the lithography bays of the semi-conductor industry, we are mainly interested in the case that there is only one resource per job. However, one might also wonder what happens if there is more than one resource needed per job. We will therefore continue by looking at instances with at most q resources per job. We introduce *partition*(q) for the β field of the scheduling problem. If *partition*(q) is in the β field, there is a collection of subsets $R = \{r^1, \ldots, r^R\}$ with $r^k \subseteq J$, where every job is contained in at most q subsets. If there exist an $r^k \in R$ such that $j, j' \in r^k$, j and j' cannot be processed at the same time.
The problem $P|\textit{partition}(\text{q}), p_j = 1|\sum_j C_j$ is a special case of $P|\text{res}\cdots, \text{types} = \mathcal{R}, p_i < p|f$ with $f \in \{\sum_j w_j C_j, \sum_j T_j, \sum_j U_j\}$. Here, s is the number of resources and there are $\mathcal{R}$ types of jobs. A type of a job j is defined as the tuple $(p_j, \mathcal{R}_1(j), \ldots, \mathcal{R}_s(j))$, where $\mathcal{R}_u(j)$ is the amount of resource u required by job j. Note that, in our case, $|R| = \mathcal{R} = s$. Brucker and Krämer [7] show that it can be solved in $O(\mathcal{R}(p+s)n^{\mathcal{R}p} + \mathcal{R}^2 p n^{\mathcal{R}(p+2)})$, resulting in the following corollary.

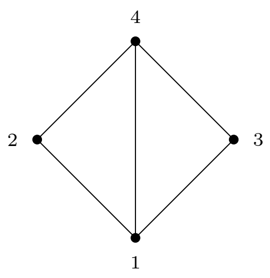
Corollary 27. $P|\textit{partition}(q), p_j = 1|\sum_j C_j$ is polynomially solvable for every $q \in \mathbb{Z}$, if the number of resources, $|R|$, is bounded.

However, we will now show that the problem becomes $\mathcal{NP}$-hard when the number of resources is not bounded, even with $q = 2$.

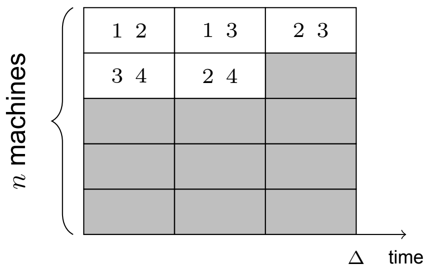
Theorem 28. $P|\textit{partition}(q), p_j = 1|\sum_j C_j$ is $\mathcal{NP}$-hard for every $q \geq 2$, if the number of machines m and resources $|R|$ are unbounded.

Proof. We will prove this by a reduction from edge coloring. In the edge coloring problem, one assigns colors (or labels) to the edges of a graph $G = (V, E)$, such that no two incident edges have the same color. Let Δ be the maximum node degree in the graph G, then Holyer [18] shows that it is $\mathcal{NP}$-hard to decide for an arbitrary graph G whether or not it can be colored using only Δ colors.
We can reduce this problem to $P|\textit{partition}(2), p_j = 1|\sum_j C_j$ as follows. Suppose we are given graph $G = (V, E)$ with $|E| = m$. Take the number of machines equal to m. Introduce a resource for every node $u \in V$, $|R| = |V|$. We also introduce a job (with $p_j = 1$) for every edge $e = \{u, v\}$ and these jobs require the resources that are associated with nodes it connects (i.e. u and v). Thus every resource will be used at most Δ times and every job uses exactly 2 resources. Lastly, we introduce $(\Delta - 1)n$ dummy jobs (with $p_j = 1$), that do no require a resource. Figure 2.9 shows an example of the reduction.

We claim that there exists an edge coloring of graph G using only Δ colors if and only if the instance of $P|\textit{partition}(2), p_j = 1|\sum_j C_j$ has an optimal value of $\frac{1}{2}\Delta(\Delta+1)m$. Suppose we are given a solution



(a) Graph $G = (V, E)$



(b) Optimal schedule for instance the instance. Dashed lined jobs represent dummy jobs and number represent the resources

Figure 2.9: Example of the reduction from edge coloring to $P|$*partition*$(2), p_j = 1|\sum_j C_j$ with a graph with $\Delta = 3$

to edge coloring problem which uses Δ colors, then we can put each color on a different time slot. So all jobs associated with an edge of the first color will be put on machines in the first time slot. We fill up all unused machine time until time Δ with dummy jobs. Since jobs only share a resource if they were incident in the graph, we will not have any resource conflict and all machines will be filled with jobs until time Δ, thus resulting in an objective value of $\frac{1}{2}\Delta(\Delta + 1)m$.
Suppose we have a solution of the above instance of $P|$*partition*$(2), p_j = 1|\sum_j C_j$ with objective value $\frac{1}{2}\Delta(\Delta + 1)m$. Then in each time slot we look for the jobs associated with a node and give them the same color. Since the objective value is $\frac{1}{2}\Delta(\Delta + 1)m$ there are no jobs after time Δ, hence there are only Δ colors. Since all jobs associated with incident edges share a resource, no two incident edges will share the same color and hence we have found an edge coloring using Δ colors. □

2.8. Conclusion

In this paper, we considered the problem of minimizing the total completion time while scheduling jobs that each use exactly one resource, $P|$*partition*$|\sum_j C_j$. Although the complexity of $P|$*partition*$|\sum_j C_j$ remains unclear, we saw that similar problems such as $P|$*partition*$, \mathcal{M}_r|\sum_j C_j$, $P|$*partition*$(2), p_j = 1|\sum_j C_j$ and $P|$*partition*, *unmovable*$, p_j = 1|\sum_j C_j$ are $\mathcal{NP}$-hard. Therefore, we conjecture that $P|$*partition*$|\sum_j C_j$ is $\mathcal{NP}$-hard as well.
The problem $P|$*partition*$|\sum_j C_j$ always has an optimal solution where jobs sharing the same resource are ordered by the processing time. Such an optimal solution might even be more structured. For example, it remains open whether or not there is always an optimal solution that yields the SPT order property on each machine for all jobs on that machine.
We showed that the SPT-*available* rule gives a $(2 - \frac{1}{m})$-approximation. This bound may not be tight. There is a lower bound of $\frac{4}{3}$ on the approximation factor. Closing this gap is another interesting open problem, as well as designing other approximation algorithms with better approximation ratios.

3
Exact Algorithms

This chapter will introduce several exact methods for $P|\textit{partition}|\sum_j C_j$. Four algorithms are based on an integer linear programming (ILP) problem, meaning the problem is modeled using only integer decision variables. The fifth algorithm is based on a mixed integer linear programming (MILP) problem, meaning the problem is modeled using decision variables of which a subset is restricted to be integer. These models can be solved using a variation of different solvers. All models, except for the model in Section 3.2, are based on other models found in the literature. Note that we do not expect solvers to be able to solve these models in polynomial time. Solving (mixed) integers linear programs is $\mathcal{NP}$-hard in general. Finally we discuss a dynamic programming algorithm.

Throughout the (Mixed) Integer Linear Programs, we will use the following notations: $\forall j$: for all jobs $j \in J$, $\forall i$: for all machines $1 \leq i \leq m$, $\forall t$: for all possible starting/ending times $t \in T$ and $\forall r^k$: for all resources $r^k \in R$.

3.1. ILP with Starting Time Variables

The first model is an integer linear program for $R|\textit{partition}|\sum_j w_j C_j$, based on the model for $R||\sum_j w_j C_j$ using time-indexed variables on starting times from [28]. The processing times p_{ij} are assumed to be integer. One additional type of constraint has been added (3.1d) for each resource $r^k \in R$, with R the set of resources. Our decision variables are as follows:

$$x_{ijt} = \begin{cases} 1, & \text{if job } j \text{ starts at time } t \text{ on machine } i \\ 0, & \text{else} \end{cases}$$

We then get the following integer linear program:

$$
\begin{aligned}
&\underset{x}{\text{minimize}} && \sum_{i,j,t} w_j \cdot x_{ijt} \cdot (t + p_{ij}) && && \text{(3.1a)}\\
&\text{s.t.} && \sum_{i,t} x_{ijt} = 1, && \forall j && \text{(3.1b)}\\
& && \sum_{j} \sum_{s=t-p_{ij}+1}^{t} x_{ijs} \leq 1 && \forall i, \forall t && \text{(3.1c)}\\
& && \sum_{j \in r^k} \sum_{i} \sum_{s=t-p_{ij}+1}^{t} x_{ijs} \leq 1 && \forall r^k, \forall t && \text{(3.1d)}\\
& && x_{ijt} \in \{0,1\} && \forall i, \forall j, \forall t && \text{(3.1e)}
\end{aligned}
$$

This model can be further simplified for $P|\textit{partition}|\sum_j C_j$ by setting $p_{ij} = p_j$ and $w_j = 1$ for all $j \in J$. One can best read the second constraint as: at time t on machine i, at most one job may be active. The third constraint is similar, but should then hold for each resource, over all machines.

The number of decision variables for this model is $m \cdot n \cdot |T|$, where T is the set of possible starting times. The size of this set is $\mathcal{O}(2^n)$ in theory, but is smaller in practice if we find an upper bound t_{max} on the possible starting times. A possible value for t_{max} could be $\sum_j p_j/m + \max_k\{q_k\}$, which is the average ending time of the machines, with the largest resource appended to it.

3.2. ILP with Preemptions

This model is inspired by Theorem 11 which states that the optimal values of $P|\textit{partition}|\sum_j C_j$ and $P|\textit{partition}, \textit{prmp}|\sum_j C_j$ are the same. Thus, one can just allow for preemptions. The decision variables are defined as follows:

$$
x_{kt} = \begin{cases} 1, & \text{if resourse } r^k \text{ is used at the interval } [t, t+1] \\ 0, & \text{else} \end{cases}
$$

$$
y_{jt} = \begin{cases} 1, & \text{if job } j \text{ finishes at time } t+1 \\ 0, & \text{else} \end{cases}
$$

Theorem 10 states that in any optimal solution of $P|\textit{partition}, \textit{prmp}|\sum_j C_j$, the jobs must be processed in order of shortest processing time per resource. Hence, if a resource has been used for a certain amount of time, we can identify which jobs have been finished. This can be done in the following way: let q_j be the cumulative processing time for job j, that means that q_j time units of that resource must have been processed, until that job is done. For example, if resource r^k has jobs of lengths $3, 4$ and 6. Then, the cumulative processing times are $3, 7$ and 13. Then we have the following integer linear program:

$$
\begin{aligned}
\underset{x,w}{\text{minimize}} \quad & \sum_{t,j} y_{jt}(t+1) && && \text{(3.2a)}\\
\text{s.t.} \quad & \sum_{k} x_{kt} \leq m, && \forall t && \text{(3.2b)}\\
& \sum_{s=0}^{t} x_{kt} \geq y_{jt} q_j, && \forall t, \forall j && \text{(3.2c)}\\
& \sum_{t} y_{jt} = 1, && \forall j && \text{(3.2d)}\\
& \sum_{t} x_{kt} = \sum_{j|r_j=r^k} p_j, && \forall r^k && \text{(3.2e)}\\
& x_{kt} \in \{0,1\} && \forall r^k, \forall t && \text{(3.2f)}\\
& y_{jt} \in \{0,1\} && \forall j, \forall t && \text{(3.2g)}
\end{aligned}
$$

Constraint (3.2b) is to make sure that there are at most m machines active at any interval. Constraint (3.2c) is to find when jobs are actually finished. Constraint (3.2e) has been added to make sure that the machines stay inactive after all jobs have finished. This is not a necessary constraint, but makes the set of optimal solutions smaller.

Note that this model may be able to be extended to $P|$*partition*, *prmp*$|\sum_j w_j C_j$, since we conjecture that the proof of Theorem 10 can be adjusted for the case with weights. The jobs are then ordered in order of shortest weighted processing time (p_j/w_j). After an optimal solution has been found for this integer linear program, the solution still needs to be transformed into a schedule without preemption. Using Theorem 11, one can construct an algorithm that removes these preemptions in a structured way.

The number of decision variables for this model is $(K+n)\cdot t_{max}$, with K the number of resources and t_{max} an upper bound on the completion times. Again, a possible value for t_{max} could be $\sum_j p_j/m + \max_j\{q_j\}$, which is the average ending time of the machines, with the largest resource appended to it.

3.3. ILP with Precedence Constraints

Theorem 12 states that all jobs are in order of shortest processing time per resource. Hence, we can describe our problem as a special case of scheduling with chain precedence constraints and use the following integer linear program for scheduling with precedence constraints ($P|$*prec*$|\sum_j w_j C_j$) from [23]. We have the following decision variables:

$$
x_{jt} = \begin{cases} 1, & \text{if job } j \text{ finishes at time } t \\ 0, & \text{else} \end{cases}
$$

We conjecture that the proof of Theorem 12 can be adjusted for the problem with weights ($P|$*partition*$|\sum_j w_j C_j$). The jobs are then ordered in shortest weighted processing time (p_j/w_j). One can simply put the weights to 1 if they are not required. We can define the following Integer Linear Program for $P|\textit{partition}|\sum_j w_j C_j$:

$$
\begin{aligned}
\underset{x}{\text{minimize}} \quad & \sum_j w_j \sum_t x_{jt} \cdot t && && \text{(3.3a)}\\
\text{s.t.} \quad & \sum_t x_{jt} = 1, && \forall j && \text{(3.3b)}\\
& \sum_{j, t \in [t', t'+p_j)} x_{jt} \leq m, && \forall t' && \text{(3.3c)}\\
& \sum_{t < t' + p_{j'}} x_{jt} \leq \sum_{t < t'} x_{j't}, && \forall j, j', t' : j \prec_P j' && \text{(3.3d)}\\
& x_{jt} = 0 && \forall j, \forall t < p_j && \text{(3.3e)}\\
& x_{jt} \in \{0, 1\} && \forall j, \forall t && \text{(3.3f)}
\end{aligned}
$$

Remember that $j \prec j'$ if j comes before j' in the list of jobs sorted on shortest (weighted) processing time and that $j \prec_P j'$ if $j \prec j'$ and j and j' use the same resource. Constraints (3.3b) and (3.3c) are to ensure that each job is scheduled exactly once, and at most m jobs are active at the same time. Notice that in Constraint (3.3d), $\sum_{t<t'+p_{j'}} x_{jt}$ will be 1 if and only if job j finishes before $t' + p_{j'}$. Hence, then job j' should finish before t'.

We can use a different set of constraints for (3.3d):

$$\sum_t t \cdot x_{j,t} + p_{j'} \leq \sum_t t \cdot x_{j',t}, \forall j, j' : j \prec_P j'$$

since $\sum_t t \cdot x_{j,t}$ is equal to the time that job j ends. Hence this new equation can be read as "job j must end at least $p_{j'}$ time units before job j' finishes if j precedes j' ".

The number of decision variables for this model is $n \cdot |T|$, where T is the set of possible completion times.

3.4. MILP with Precedence Constraints

This model is based on the model from van den Bogaerdt [27]. The original model has a combination of precedence constraints and partition constraints, as well as additional release dates for each partition and due dates for each job. One can put all release and due dates to 0 to simplify to model. We also used Theorem 12 to simplify the partition constraints, since we know in which order the jobs must be processed per partition (or resource). This way, the partition constraints become precedence constraints in the model.

$$
\begin{aligned}
C_j &= \text{ is the completion time of job } j\\
Y_{i,j} &= \begin{cases} 1, & \text{if job } j \text{ is scheduled on machine } i\\ 0, & \text{else} \end{cases}\\
X_{j,j'} &= \begin{cases} 1, & \text{if job } j \text{ is scheduled before job } j' \text{on the same machine}\\ 0, & \text{else} \end{cases}
\end{aligned}
$$

Then we can define the following Mixed Integer Linear Program for our problem:

$$
\begin{aligned}
&\underset{x}{\text{minimize}} && \sum_j C_j && && \text{(3.4a)}\\
&\text{s.t.} && \sum_i Y_{i,j} = 1, && \forall j && \text{(3.4b)}\\
& && Y_{i,j} + \sum_{i' \neq i} Y_{i',j'} + X_{j,j'} \leq 2 && \forall i, \forall j \prec j' && \text{(3.4c)}\\
& && C_{j'} - C_j + L(3 - X_{j,j'} - Y_{i,j} - Y_{i,j'}) \geq p_j && \forall i, \forall j \prec j' && \text{(3.4d)}\\
& && C_j - C_{j'} + L(2 - X_{j',j} - Y_{i,j} - Y_{i,j'}) \geq p_{j'} && \forall i, \forall j \prec j' && \text{(3.4e)}\\
& && C_{j'} - C_j \geq p_{j'} && \forall j \prec_P j' && \text{(3.4f)}\\
& && C_j \geq p_j && \forall j && \text{(3.4g)}\\
& && Y_{i,j} \in \{0,1\} && \forall i, \forall j && \text{(3.4h)}\\
& && X_{j,j'} \in \{0,1\} && \forall j, j' && \text{(3.4i)}
\end{aligned}
$$

Remember that $j \prec j'$ if j comes before j' in the list of jobs sorted on processing time and that $j \prec_P j'$ if $j \prec j'$ and j and j' use the same resource. Constraint (3.4b) makes sure each job is scheduled exactly once. Constraint (3.4c) states that $X_{j,j'} = 0$ if j and j' are scheduled on different machines. In the third constraint, a large constant L is being used. This should be a large number, such that the constraint is only enforced if L is multiplied with 0. For example, in the case of Constraint (3.4d), $C_{j'} - C_j \geq p_j$ only needs to hold if $X_{j,j'} = Y_{i,j} = Y_{i,j'} = 1$, so j is scheduled before j' on the same machine i. L should be an upper bound on $C_{j'} - C_j + p_j$, hence we choose $L = \sum_j p_j - \min_j\{p_j\} + \max_j\{p_j\}$. Constraint (3.4e) is similar to (3.4d), but is enforced if j is scheduled *after* j' on the same machine i. Constraint (3.4f) models our partition constraint as precedence constraints.

The number of decision variables for this model is $\mathcal{O}(\frac{1}{2}n^2 + n \cdot m)$.

3.5. Dynamic Programming

The exact solution can also be computed using dynamic programming. In the algorithm, the jobs are sorted in order of processing time per resource. The jobs are then scheduled one by one iteratively. The next job to be scheduled has to be the shortest job of one of the K resources, resulting in a maximum of K subproblems per iteration. Hence it has a runtime of $\mathcal{O}(K^n)$.

4 Heuristics

In this chapter, several different heuristics for $P|partition|\sum_j C_j$ are described. In practice, it may be useful to use a heuristic instead of an exact algorithm, since the runtime of the exact algorithms from Chapter 3 are expected to be exponential in terms of the input size. First we will look at the SPT-*available* rule and an algorithm which is closely related to it, called semi-SPT-*available*. The next two heuristics are based on dynamic programming. Although the runtimes of those algorithms are also exponential in terms input size, it is interesting to see how well they operate, and until which size of input they may be a good alternative. Finally, we will look at heuristics based on LP relaxations.

4.1. SPT-*available*

SPT-*available* is the greedy algorithm already discussed in Section 2.4. In summary, the algorithm sorts the list of jobs in order of processing time and then schedules each job at the first available time. Remember that Section 2.4 states that it gives a $(2-\frac{1}{m})$-approximation and it cannot give an α-approximation with $\alpha < \frac{4}{3}$. It will be interesting to inspect the accuracy of SPT-*available* for differently distributed data, as it may depend greatly on the type of instances. The runtime of this algorithm is $\mathcal{O}(n \log n)$, since the jobs must be sorted in order of shortest processing time.

4.2. Semi-SPT-*available*

Semi-SPT-*available* is an algorithm, designed with SPT-*available* and its limitations in mind. As seen in Example 1, SPT can perform poorly if an instance contains several longer jobs using the same resource. The new algorithm should prevent scheduling all jobs of that resource at the end. To do this, it keeps track of how much time a resource should still be scheduled to be used, we call this P_k for each resource r^k. If at any time the largest-to-be-scheduled resource will finish late ($P_k + t > \sum_j p_j/m$ with t the first possible starting time for the to be scheduled jobs), all jobs of that resource will be scheduled at the first possibility. Otherwise, schedule the job with the shortest processing time at the earliest possibility. The runtime is $\mathcal{O}(n \log n)$, due to the sorting of the jobs.

4.3. Dynamic Programming Heuristics

The two following heuristics are based on the exact dynamic programming algorithm from Section 3.5, but allow substantially fewer subproblems. The first difference is that we only allow solutions where the jobs are also ordered in processing time per machine. This is because we conjecture that there is always an optimal solution, with jobs sorted per machine. Furthermore, instead of creating (a maximum of) K subproblems, we only allow the creation of two kinds of subproblems, inspired by the algorithms SPT-*available* and Semi-SPT-*available*:

- scheduling of the job with the shortest processing time (SPT order)
- scheduling of the job with the shortest processing time of the resource which has the largest amount of time yet to be processed from all resources that are allowed to be scheduled at that time.

This makes the running time of the heuristic of order $\mathcal{O}(2^n)$. This is still exponential, but can already be used for larger instances than the exact dynamic programming method.

The second heuristic reduces the amount of subproblems even further by creating the second set of subproblems only if that resource will finish late, i.e. $P_k + t > \sum_j p_j/m$.

Notice that the subproblems seem to be unique quite often, since each subproblem depends on the already scheduled jobs and the times the machines and the resources become available. Hence, the dynamic programming part of this algorithm will not be used very often, as not a lot of the subproblems overlap. We chose to still incorporate the dynamic programming function instead of creating a new recursive function, since the dynamic programming part does not consume a lot of runtime and memory, but can help stabilize the function for big numbers. However, recursive functions are expected to have similar results.

4.4. LP Relaxations

In an LP relaxation, one solves the (mixed) integer program, but without the integrality restrictions. The (non-integer) solution of this linear program is then used to create a solution for the original problem. The conversion used for our LP relaxations creates a list of the jobs sorted on their average starting times and then schedules the jobs at the first possibility in that order. The choice for average starting times is based on the fact that any optimal integer solution would remain optimal after the conversion. The average starting would then simply be equal to the starting time of a job, and rescheduling the jobs in this order will give the exact same schedule. This is in contrast with sorting on average *completion* time for example. In Figure 4.1 one can see that an optimal solution sorted and then rescheduled on completion time is not optimal.

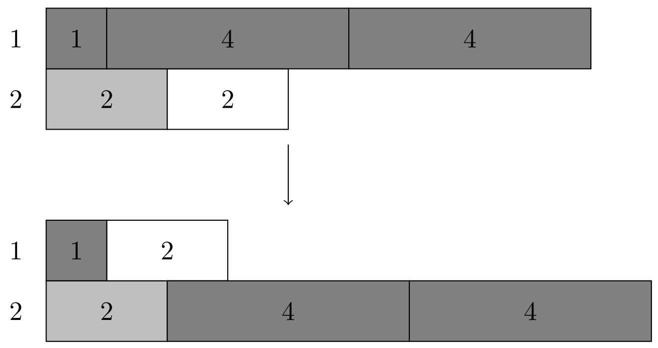


Figure 4.1: Example of rescheduling optimal solution in order of completion time does not give optimal solution.

Although the relaxation method should be polynomial, assuming that the processing times are bounded, the number of variables becomes large quite fast.

5 Experimental Results

In this chapter, the proposed exact algorithms and heuristics will be assessed. This will be done using two different kinds of data: uniformly distributed processing times and processing times based on the semiconductor industry. The algorithms will be evaluated based on their average runtimes, the size of instance they can handle and their accuracy.

5.1. Implementation

MATLAB was used for the implementation of our algorithms together with opti toolbox [12] for solving the (mixed) integer linear programs for the exact algorithms and the linear programs for some of the heuristics. The opti toolbox would automatically select the solver for each instance for each linear program. For (Mixed) Integer Linear Programming, opti toolbox chooses from the following four solvers:

- CBC (https://projects.coin-or.org/Cbc/),
- GLPK (http://www.gnu.org/software/glpk/),
- LP_SOLVE [3] and
- SCIP [1].

For Linear Programming, opti toolbox chooses from the following seven solvers:

- CLP (https://projects.coin-or.org/Clp),
- CSDP [6],
- CSDP [2],
- GLPK (http://www.gnu.org/software/glpk/),
- LP_SOLVE [3],
- OOQP [16] and
- SCIP [1].

During the testing, the choice for the usage of solver was left to the opti toolbox, as the impact of the usage of different solvers was not studied.

In all the linear programs, sparse matrices were used to speed up the process and also to significantly decrease the required memory space.

5.2. Test Data

The data our algorithms have been tested on can be divided into two different categories. The first category is called *uniform*, as the processing times of the jobs are uniformly distributed. For input values m, K, n and p_{max}, instances are created that use all K resources and that have processing times $p_j \in \mathbb{N}$ taken from a uniform distribution going from 1 to p_{max}. The second category is based on the test data of [4], also researching the semiconductor industry. For input values m, K and n, instances are created in the following way:

- Processing time per wafer for resources are uniformly random generated on the interval $[5, 10]$.
- For the first K jobs, the resource for the jth job is the jth resource. The remaining jobs have a randomly selected resource.
- With a probability of 0.9, the number of wafers for a job is 25. For the other jobs, the number of wafers are uniformly random generated on the interval $[1, 25]$.
- The processing time of each job is now determined by the processing time per wafer of its resource times the number of wafers.

We will also refer to this kind of data as *non-uniform*.

5.3. Exact Algorithms Results

The following algorithms were tested:

- **ILP start var**, which is the algorithm solving the integer linear program based on variables indicating the starting times of jobs at machines from Section 3.1.
- **ILP preempt**, which is the algorithm solving the integer linear program allowing preemptions from Section 3.2.
- **ILP prec v1**, which is the algorithm solving the integer linear program with precedence constraints from Section 3.3, using the first variant for constraint (3.3d).
- **ILP prec v2**, which is the algorithm solving the integer linear program with precedence constraints from Section 3.3, using the second variant for constraint (3.3d).
- **MILP**, which is the algorithm solving the mixed integer linear program with precedence constraints from Section 3.4.
- **Dynamic**, which is the dynamic programming algorithm from Section 3.5.

All the tests on the exact algorithms were done 50 times per set of input variables, to decrease the variance of the found averages. In the figures, one may see dotted lines in the charts. These dotted lines represent that the measured results are not to be trusted, but may give an indication on how the algorithm would perform. In most of these cases, the algorithm stopped working due to errors like maximum nodes reached. The maximum on the number of nodes was chosen automatically by the solvers. This value could have been chosen differently, but it was chosen not to do so. This was since the computer sometimes already was overloaded during the testing with the current maximum number of nodes, meaning that the computer became temporarily unresponsive or sometimes even crashed completely. By using a computer with better hardware, using different (commercial) solvers and/or by allowing more iterations, one might then be able to solve the problems for bigger values of n. Note that if a line stops at a certain n, we stopped testing that algorithm because of earlier results.

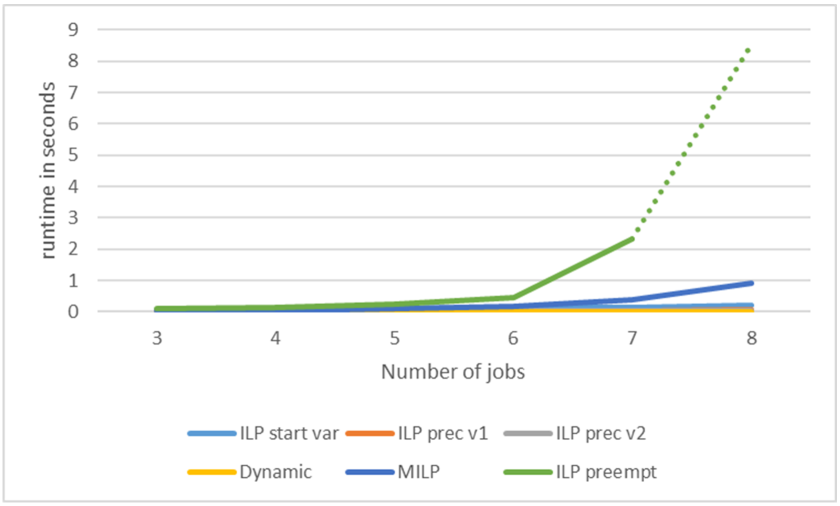


Figure 5.1: Average running time for $m = 2$, $K = 3$ and $p_{max} = 5$ for the exact algorithms. Dotted lines represent that the algorithm did not always succeed in optimizing and thus the measured results are not to be trusted, but may give an indication on how the algorithm would perform.

ILP with Preemptions

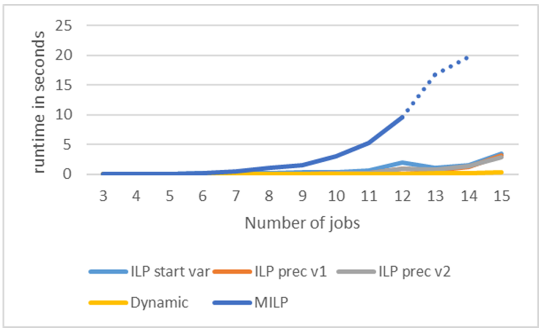
This algorithm performed poorly. In Figure 5.1, one can see that even for small problems, the algorithm is already quite slow compared to the other exact algorithms. Note that the algorithm didn't always fully optimize the integer linear program for $n \geq 8$, due to maximum nodes reached. The bad performance, compared to the other algorithms, is probably due to the great amount of decision variables, combined with the fact that there are a lot of optimal solutions. Because of this, we decided to not further test this algorithm.

MILP

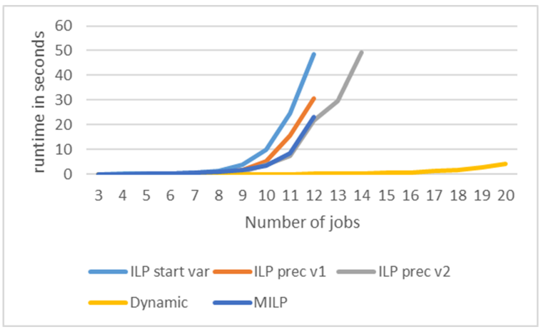
The MILP model didn't perform that well either. Although it has a similar running time for $m = 2$, $K = 3$, $p_{max} = 250$ compared to the other algorithms, it has large runtimes for almost all other input settings. For example, in Figure 5.2 one can see the difference between uniform and non-uniform for $m = 2$, $K = 3$. We also included the results for $m = 3$, $K = 3$, $p_{max} = 250$, for which one can notice that the solver did not always find optimal values for the instance with only eight jobs.

Other Algorithms and Overall Performance

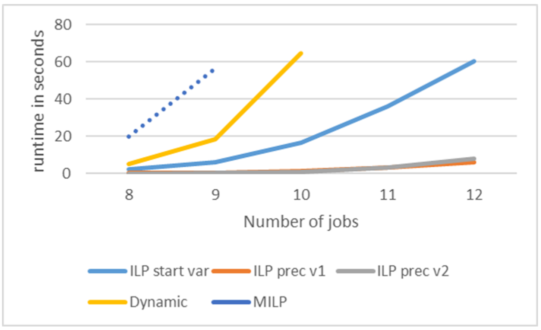
We continued testing the other algorithms. The results of this can be found in Figure 5.3. For the tests, we kept increasing n until the computer became unresponsive because MATLAB stopped working and crashed. Note that the algorithms based on (Mixed) Integer Linear Programs perform better on the uniform data set. This is mainly due to the fact that p_{max} is smaller, and thus the amount of decision variables is smaller. One can see clearly that the runtime of the dynamic programming algorithm strongly depends on the number of resources. For $K = 3$, the algorithm is the fastest of all algorithms up until 20 jobs. But for $K = 4$, the other methods seem to be a lot faster. Overall, the exact algorithms are not able to solve big instances. From these tests, it seems that ILP prec v2 is the best overall choice for exact algorithm, since it performs well for most tested inputs. However, the algorithms might not be suitable for use in practical settings. For example, in a semiconductor fabrication plant, there could be 37 machines and 700 jobs that are to be scheduled. These exact algorithms with the current solvers are not able to solve the instances.



(a) $m = 2$, $K = 3$, non-uniform

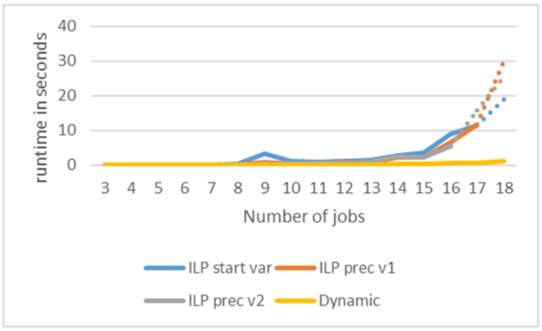


(b) $m = 2$, $K = 3$, $p_{max} = 250$

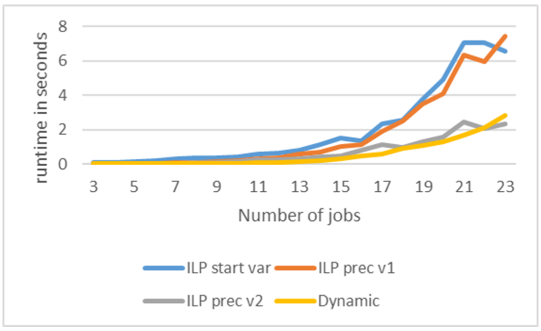


(c) $m = 3$, $K = 8$, $p_{max} = 250$

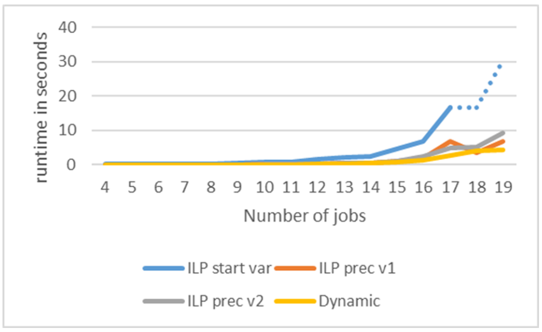
Figure 5.2: Average running time of tested exact algorithms. Dotted lines represent that the algorithm did not always succeed in optimizing and thus the measured results are not to be trusted, but may give an indication on how the algorithm would perform.



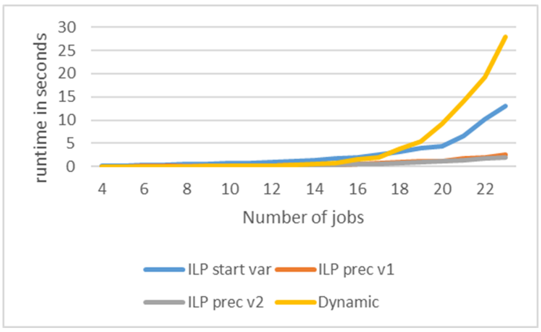
(a) $m = 2$, $K = 3$, non-uniform



(b) $m = 2$, $K = 3$, $p_{max} = 25$



(c) $m = 3$, $K = 4$, non-uniform



(d) $m = 3$, $K = 4$, $p_{max} = 25$

Figure 5.3: Average running time of tested exact algorithms. Dotted lines represent that the algorithm did not always succeed in optimizing and thus the measured results are not to be trusted, but may give an indication on how the algorithm would perform.

5.4. Heuristics Results

The following algorithms were tested:

- **SPT**, which is the algorithm called SPT-*available*, scheduling the jobs in order of shortest processing time, as explained in Section 4.1.
- **Semi SPT**, which is the algorithm called semi-SPT-*available*, scheduling the jobs in order of shortest processing time, unless a resource is already ending after $\sum_j C_j/m$ as explained in Section 4.2.
- **Dynamic 1**, which is the using dynamic programming creating only 2 subproblems each iteration as explained in Section 4.3.
- **Dynamic 2**, which is a slight modification to Dynamic 1, creating even less subproblems as explained in Section 4.3.
- **LP Relax 1**, which creates a solution based on a relaxation of the *first* version of the integer linear program with precedence constraints from Section 3.3. The method to convert its solution into an integer solution is explained in Section 4.4.
- **LP Relax 2**, which creates a solution based on a relaxation of the *second* version of the integer linear program with precedence constraints from Section 3.3. The method to convert its solution into an integer solution is explained in Section 4.4.

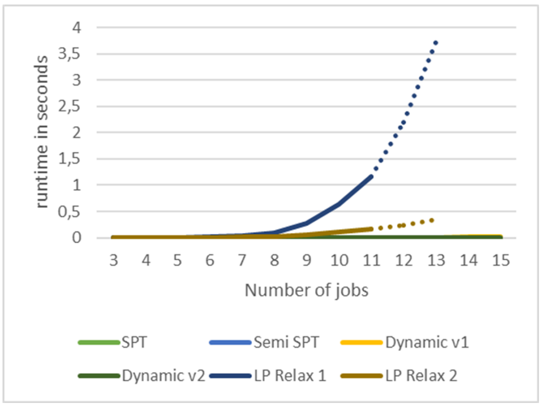
All the tests on the heuristics were done 100 times per set of input variables, to decrease the variance of the found averages. In the figures, one may see dotted lines in the charts. These dotted lines represent that the measured results are not to be trusted, but may give an indication on how the algorithm would perform. In most of these cases, the algorithm stopped working due to errors like maximum nodes reached. By using a computer with more space and/or by allowing more iterations, one might then be able to solve the problems for bigger values of n.

Redefining Accuracy Ratio

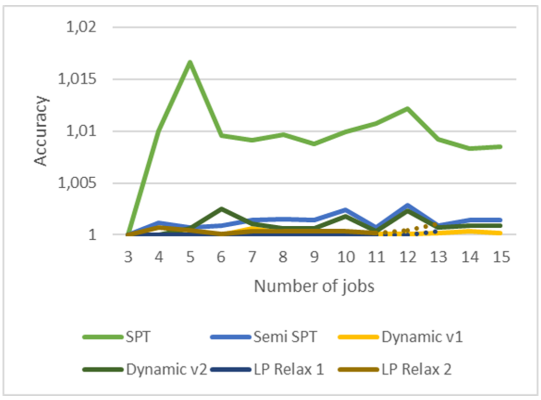
The quality of a heuristic mainly depends on the running time and the accuracy. The running time will be measured as the average running time. One could measure the accuracy as the average performance ratio, i.e. the ratio between the found solution and the optimal solution. However, an optimal solution is not always available as our exact algorithms don't work sufficiently. Hence we will have to choose a different measure. One option would be to use a lower bound om the optimal solution. However, the only known lower bounds are the optimal value of $P||\sum_j C_j$ and relaxations of the (mixed) integer linear programs. The optimal value of $P||\sum_j C_j$ is not a good choice, since the lower bound is too low. More often than not, the optimal values differ with a factor bigger than 2. However, the relaxations are not robust and stop working for already quite small instances. Hence, we propose the following measure: the ratio between the found solution and the best found solution by any of the heuristics. This may not always be equal to the average performance ratio, but tests up until $n = 20$ show that it seems to be almost always equal to the performance ratio in the tests using all six heuristics. Remember though, that the quality of this measure becomes less reliable as the number of heuristics one tests simultaneously decreases. Nevertheless, it remains a good way to measure the difference in accuracy between the heuristics.

Relaxations

The ILP relaxation stopped working quite fast. For example, in Figure 5.4, one can see that the algorithms might be interrupted early at just 11 jobs. This was mostly due to errors such as maximum nodes reached. This resulted not only in non-optimal solutions, but also in large running times. Probably, the average accuracy ratio will be a bit better than the dotted lines in Figure 5.4b. This is because the solutions are based on the non-optimal solutions from the relaxations. For big n, the relaxation algorithms quickly became the slowest heuristics. We decided to stop testing the algorithms as they were clearly not suitable for use in large scale problems.



(a) Average run-time.



(b) Average accuracy ratio.

Figure 5.4: Experimental results for all heuristics $m = 2$, $K = 3$, $p_{max} = 100$. Dotted lines represent that the algorithm did not always succeed in optimizing and thus the measured results are not to be trusted, but may give an indication on how the algorithm would perform.

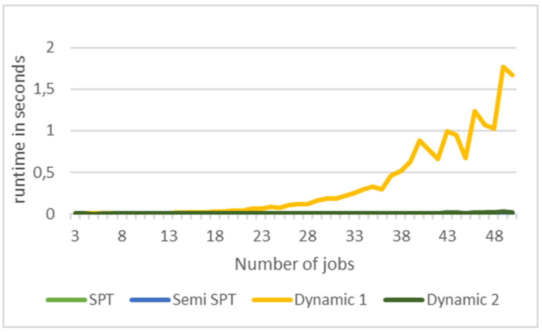
Runtime

We will further investigate the quality of the other four heuristics. These were tested on the settings $(m = 2, K = 3)$, $(m = 3, K = 4)$ and $(m = 3, K = 8)$, all for non-uniformly and uniformly distributed processing times. The p_{max} was chosen to be equal to 250, as the maximal processing time in the non-uniformly case is also 250. The first thing we notice is that *dynamic 1* is by far the slowest algorithm. This is because it creates a (possible) exponential number of subproblems. The average runtime of dynamic appears to become larger as K increases. This can be explained as follows. If the smallest job uses the biggest resource, only one subproblem will be created. If K increases, the chance of this happening decreases. Hence, the dynamic 1 function actually becomes more and more exponential. All other algorithms processed everything within a fraction of a second. As expected though, dynamic 2 is slightly slower than SPT and Semi-SPT. The greedy algorithms executed everything in 2 milliseconds, whereas dynamic 2 sometimes took sometimes a tenth of a second. With n becoming even bigger, this difference may become bigger.

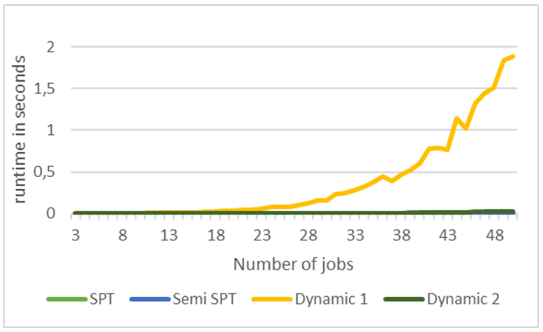
Accuracy

The results are displayed in Figure 5.6. Remember that the accuracy is measured as the ratio between the found value and the best found value. As expected, the accuracy of the algorithms depends on the type of data. For example, SPT clearly performs better for uniformly distributed processing times. This could be explained in the following way. In the non-uniformly data, if a resource has a long processing time, there is a big chance that all its jobs also have long processing times. In this case the SPT algorithm will put these long jobs at the end of the schedule, just like in Example 1. In the case of uniformly distributed processing times, such situation will not happen as often.

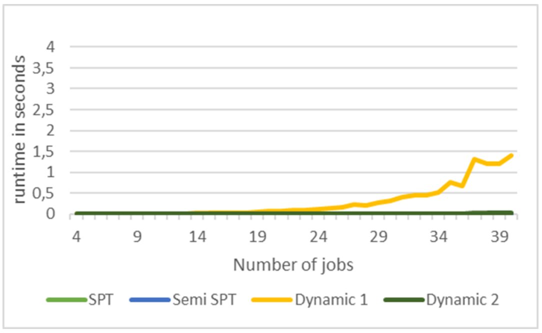
As K increases, the accuracy of SPT and Semi-SPT becomes better. This could be due to the fact that SPT seems to perform better as K gets bigger, since the chance that a resource is needed at the same time for another job becomes smaller. Also, the Semi-SPT algorithm will look more like SPT, since the chance that a color is big enough to end after the threshold becomes smaller. In all cases, the dynamic functions seem to outperform the greedy algorithms SPT and Semi-SPT. This is logical



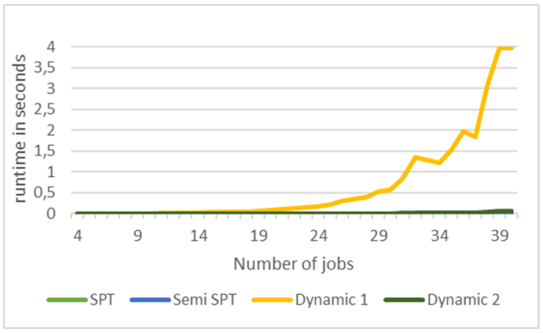
(a) $m = 2$, $K = 3$, non-uniform



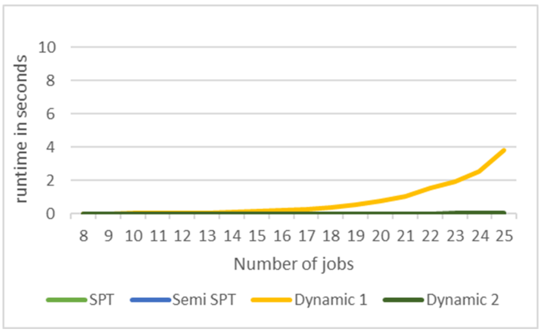
(b) $m = 2$, $K = 3$, $p_{max} = 250$



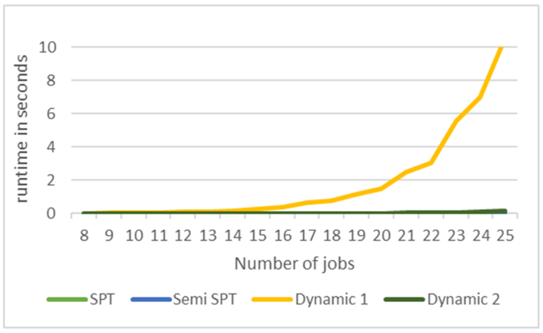
(c) $m = 3$, $K = 4$, non-uniform



(d) $m = 3$, $K = 4$, $p_{max} = 250$



(e) $m = 3$, $K = 8$, non-uniform



(f) $m = 3$, $K = 8$, $p_{max} = 250$

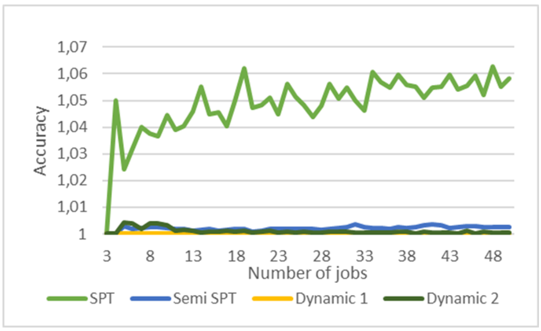
Figure 5.5: Average running time of best tested heuristics.

since they check more possible solutions. For example, SPT is always a subproblem of the dynamic functions.

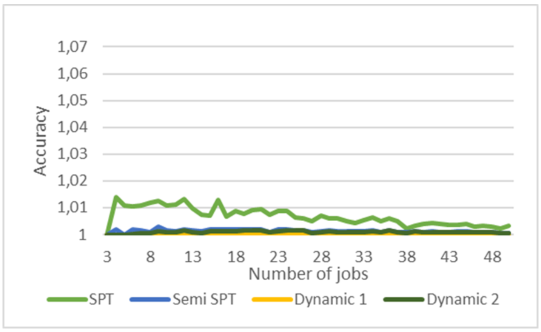
In Table 5.1 one can see the maximal found accuracy ratio in all executed tests. Here, one can see that although the average accuracy ratios of Semi-SPT, Dynamic 1 and dynamic 2 are all under 1.01, the algorithms can give a solution of at least a 12.5%, 4.9% and 12.5% bigger than the optimal solution.

m	K	pmax	SPT	Semi SPT	Dyn 1	Dyn 2	LPHeur2	LPheur3
2	3	25	1,179	1,125	1,021	1,125	1,006	1,109
2	3	100	1,161	1,068	1,049	1,116	1,000	1,038
2	3	250	1,157	1,119	1,000	1,062		
2	3	non-uniform	1,258	1,087	1,009	1,087	1,023	1,045
3	4	250	1,148	1,059	1,000	1,059		
3	4	non-uniform	1,235	1,115	1,010	1,103		
3	8	250	1,092	1,063	1,020	1,037		
3	8	non-uniform	1,163	1,049	1,000	1,048		
		Overall maximum	1,258	1,125	1,049	1,125	1,023	1,109

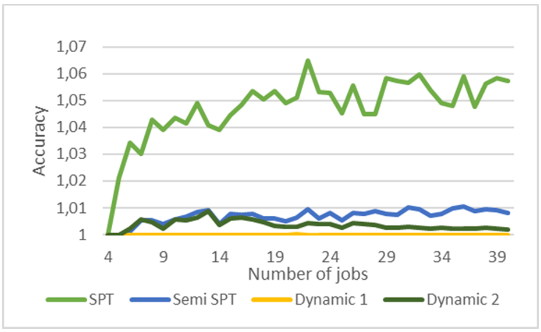
Table 5.1: The maximal found accuracy ratio, which is a lowerbound on the approximation ratio



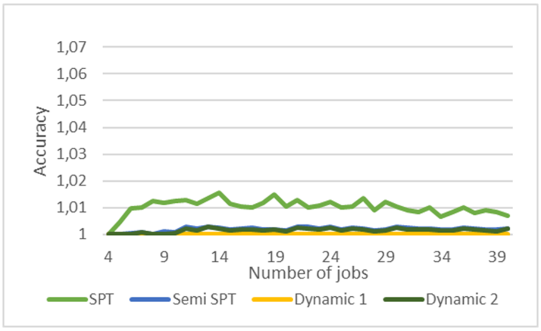
(a) $m = 2,\ K = 3$, non-uniform



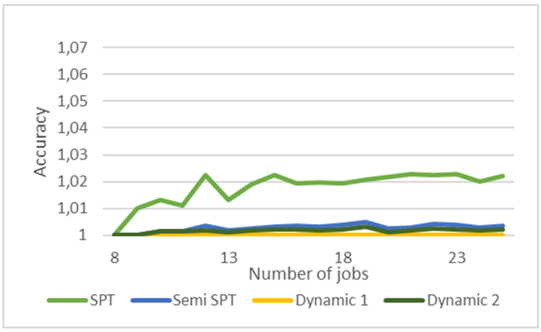
(b) $m = 2,\ K = 3,\ p_{max} = 250$



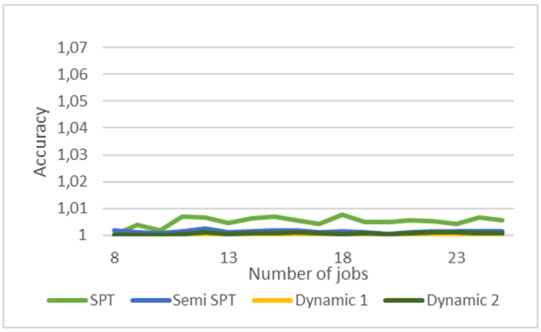
(c) $m = 3,\ K = 4$, non-uniform



(d) $m = 3,\ K = 4,\ p_{max} = 250$



(e) $m = 3,\ K = 8$, non-uniform



(f) $m = 3,\ K = 8,\ p_{max} = 250$

Figure 5.6: Average accuracy ratio of best tested heuristics.

5.5. Conclusion

The exact algorithms seem too slow for use in practice with big instances. If an exact algorithm would be used, we would recommend using the second variant of the integer linear program with precedence constraints from Section 3.3. This is because this algorithms seems to perform better for both data sets. However, different applications may lead to differently distributed data. Hence, other models may perform better depending on the application.

We would not recommend using a relaxation of an integer linear program as a heuristic as they are not able to solve for big instances. The use of the other algorithms depends on the application. We would recommend semi-SPT over SPT, since the runtimes are similar, but the accuracy of semi-SPT outperforms SPT. However, if one searches for nearly-optimal solutions, we would recommend using Dynamic 1 (which is more accurate) or Dynamic 2 which is faster), depending on the time the algorithm may take. As with most heuristics, testing is required for each application to find the most suited heuristic.

6 Conclusion and Future Work

6.1. Conclusion

This thesis is about the scheduling problem denoted by $P|\textit{partition}|\sum_j C_j$. Although the complexity of the problem has not yet been determined, we succeeded in finding important properties of the problem. Examples of this are the fact that allowing preemptions does not change the problem and the fact that all jobs must be in order of processing time per resource in an optimal solution. These theorems help us to understand the problem and may help to come up with new bounds on approximation algorithms or even new proofs about its complexity. A corollary of these theorems is that $P|\textit{partition}|\sum_j C_j$ is a special case of $P|\textit{chains}|\sum_j C_j$. Hence, algorithms for this problem could be used on our problem.

Furthermore, a greedy algorithm called SPT-*available* has been studied. Although there are examples of instances where it is not optimal (with a ratio of $\frac{4}{3}$), we also found an upper bound of $2(1-\frac{1}{m})$ on its approximation ratio.

Some related problems have been studied, such as the problem with machine subset constraints. It was shown that this problem is $\mathcal{NP}$-hard in the strong sense. As a corollary, the problem found in the semi conductor industry having unrelated machines is also $\mathcal{NP}$-hard.

Next, we created different ILPs for our problem. These exact methods are restricted to small instances because for larger instances the number of variable becomes to large. The methods may be able to solve for larger instances when using other hardware and/or different solvers. Overall, the second version of the ILP model using precedence constraints from Section 3.3 performed the best.

Finally, we studied several heuristics for the problem. Next to the two greedy algorithms SPT-*available* and semi-SPT-*available*, two dynamic programming solutions and two heuristics based on relaxations were created. If one would want to use heuristics, several heuristics should be tested, as the accuracy (and sometimes even the runtimes) may depend on the type of instances.

The heuristics based on relaxations are not recommended for use, due to the fact that they could not solve the medium sized instances. The fastest heuristics were SPT-*available* and Semi-SPT-*available*, followed by the second dynamic programming algorithm. The first dynamic programming algorithm has exponential runtime, meaning that it has long runtimes for large instances.

The first dynamic programming algorithm has the best accuracy of the tested heuristics, followed by the second dynamic programming algorithm. Semi-SPT-*available* seems to have a better approximation ratio than SPT-*available*, since the worst approximation ratio encountered in the experiments was 1.258 for SPT-*available* versus 1.125 for Semi-SPT-*available*. Hence we would advice to use Semi-

SPT-*available* over SPT-*available* in applications which desire a quick heuristic for big instances. In applications where one would like to focus more on the accuracy of the heuristic, we would recommend using Dynamic 1 (which is more accurate) or Dynamic 2 which is faster), depending on the time the heuristic may take.

6.2. Future Work

One conjecture which has not been proved is that there is always a solution with all jobs ordered in processing times per machine. I have a partial proof (not included in the thesis) which excludes many possible situations, but it still does not cover all cases. Although this conjecture may not have a direct impact on the solving of the problem, it may help in designing new (approximation) algorithms or proving their bounds.

We also conjecture that the proof of Theorem 12 can be adjusted to proof that in an optimal solution for $P|\textit{partition}|\sum_j w_jC_j$ the jobs are ordered in order of weighted completion time (p_j/w_j) per resource.

The problem 3-PARTITION was used for the reduction of $P|\mathcal{M}_j, \textit{partition}|\sum_j C_j$. This $\mathcal{NP}$-complete problem has been mentioned in some other reductions for related problems and may be the key to proving our partition constraint problem with parallel machines to be $\mathcal{NP}$-hard. It would be interesting if one could combine the proofs of $P|\textit{chains}|\sum_j C_j$ and $P|\textit{partition}, \mathcal{M}_j|\sum_j C_j$ (both having a reduction from 3-PARTITION) for proving this.

An interesting observation is that the relaxations of some of the integer linear programs seem to give the same optimal value as the corresponding integer linear program. It would be interesting to test this more intensively and see whether a good conversion algorithm can be created to create integer solutions from the non-integer optimal solution of the relaxations. One would then be able to create an approximation algorithm, or maybe even create an exact pseudo-polynomial-time algorithm. If the latter would be possible, one would have a proof for the problem being weakly $\mathcal{NP}$-hard.

It would be interesting to find a bound on the approximation ratio for the Semi-SPT-*available* algorithm or develop examples in which it does not perform well.

The choice for solvers of the (M)ILP's and LP's has been done by opti toolbox. However, choosing different solvers may effect the experimental results. It would be interesting to test different solvers, maybe even some commercial solvers, to see whether the runtimes of the methods become better.

The approximation ratios of the heuristics could only be approximated, since there was no sufficient lower bound on the optimum. Hence, there were only conclusions based on relative accuracy. Finding a better lower bound would give better insight in the actual performance of the heuristics. It may be interesting to try to bound the order of the runtime of the second dynamic algorithm in something other than 2^n. The algorithm clearly creates only one subproblem during most of its iterations and if a second subproblem is created, its size is remarkably smaller most of the time.

Finally, most of the research in this thesis focuses on the problem with parallel machines. However, it would be interesting to also develop exact algorithms and heuristics for the problem with unrelated machines.

Bibliography

[1] Tobias Achterberg. Scip: solving constraint integer programs. *Mathematical Programming Computation*, 1(1):1–41, Jul 2009. ISSN 1867-2957. doi: 10.1007/s12532-008-0001-1. URL https://doi.org/10.1007/s12532-008-0001-1.

[2] Steven J Benson, Yinyu Ye, and Xiong Zhang. Solving large-scale sparse semidefinite programs for combinatorial optimization. *SIAM Journal on Optimization*, 10(2):443–461, 2000.

[3] Michel Berkelaar, Kjell Eikland, Peter Notebaert, et al. lpsolve: Open source (mixed-integer) linear programming system. *Eindhoven U. of Technology*, 2004.

[4] Abdoul Bitar, Stéphane Dauzère-Pérès, Claude Yugma, and Renaud Roussel. A memetic algorithm to solve an unrelated parallel machine scheduling problem with auxiliary resources in semiconductor manufacturing. *Journal of Scheduling*, 19(4):367–376, 2016.

[5] Jacek Blazewicz, Jan Karel Lenstra, and AHG Rinnooy Kan. Scheduling subject to resource constraints : Classification and complexity. *Discrete Applied Mathematics*, 5:11–24, 1983.

[6] Brian Borchers. Csdp, ac library for semidefinite programming. *Optimization methods and Software*, 11(1-4):613–623, 1999.

[7] Peter Brucker and Andreas Krämer. Polynomial algorithms for resource-constrained and multiprocessor task scheduling problems. *European Journal of Operational Research*, 90(2):214–226, 1996.

[8] Peter Brucker, Bernd Jurisch, and Andreas Krämer. Complexity of scheduling problems with multipurpose machines. *Annals of Operations Research*, 70:57–73, 1997.

[9] James Bruno, Edward G Coffman Jr, and Ravi Sethi. Scheduling independent tasks to reduce mean finishing time. *Commun. ACM*, 17(7):382–387, 1974.

[10] Chandra Chekuri, Rajeev Motwani, Balas Natarajan, and Clifford Stein. Approximation techniques for average completion time scheduling. *SIAM Journal on Computing*, 31(1):146–166, 2001.

[11] Richard Walter Conway, William L Maxwell, and Louis W Miller. *Theory of scheduling*. Addison-Wesley, 1967.

[12] Jonathan Currie, David I Wilson, et al. Opti: lowering the barrier between open source optimizers and the industrial matlab user. *Foundations of computer-aided process operations*, 24:32, 2012.

[13] Jianzhong Du, Joseph YT Leung, and Gilbert H Young. Scheduling chain-structured tasks to minimize makespan and mean flow time. *Information and Computation*, 92(2):219–236, 1991.

[14] Michael R Garey and David S. Johnson. Complexity results for multiprocessor scheduling under resource constraints. *SIAM Journal on Computing*, 4(4):397–411, 1975.

[15] Michael R Garey and David S Johnson. *Computers and intractability: a guide to NP-completeness*. WH Freeman and Company, San Francisco, 1979.

[16] E Michael Gertz and Stephen J Wright. Object-oriented software for quadratic programming. *ACM Transactions on Mathematical Software (TOMS)*, 29(1):58–81, 2003.

[17] Ronald L Graham, Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling, a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

[18] Ian Holyer. The NP-completeness of edge-coloring. *SIAM Journal on computing*, 10(4):718–720, 1981.

[19] WA Horn. Minimizing average flow time with parallel machines. *Operations Research*, 21(3): 846–847, 1973.

[20] Teun Janssen, Céline Swennenhuis, Abdoul Bitar, Thomas Bosman, Dion Gijswijt, Leo van Iersel, Steèphane Dauzére-Pérès, and Claude Yugma. Parallel machine scheduling with a single resource per job. arXiv:1809.05009, 2018.

[21] Jan Karel Lenstra, AHG Rinnooy Kan, and Peter Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977. ISSN 0167-5060. doi: 10.1016/S0167-5060(08)70743-X.

[22] Joseph Y-T Leung and Chung-Lun Li. Scheduling with processing set restrictions: A survey. *International Journal of Production Economics*, 116(2):251–262, 2008.

[23] Shi Li. Scheduling to minimize total weighted completion time via time-indexed linear programming relaxations. *arXiv preprint arXiv:1707.08039*, 2017.

[24] Robert McNaughton. Scheduling with deadlines and loss functions. *Management Science*, 6(1): 1–12, 1959.

[25] René Sitters. Two NP-hardness results for preemptive minsum scheduling of unrelated parallel machines. In *International Conference on Integer Programming and Combinatorial Optimization*, pages 396–405. Springer, 2001.

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

[27] Pim van den Bogaerdt. Multi-machine scheduling lower bounds using decision diagrams. Master's thesis, Delft University of Technology, 2018. https://repository.tudelft.nl/.

[28] Tjark Vredeveld and Cor Hurkens. Experimental comparison of approximation algorithms for scheduling unrelated parallel machines. *INFORMS Journal on Computing*, 14(2):175–189, 2002.