Radboud Universiteit Nijmegen

Master Thesis Mathematics

Scheduling with job dependent machine speed

Author:
Veerle Timmermans

Supervisor:
Tjark Vredeveld
Wieb Bosma

May 21, 2014

Abstract

The power consumption rate of computing devices has seen an enormous increase over the last decades. Therefore computer systems must make a trade-off between performance and energy usage. This observation has led to speed scaling, a technique that adapts the speed of the system to balance energy and performance. Fundamentally, when implementing speed scaling, an algorithm must make two decisions at each time: (i) a scheduling policy decides which jobs to serve, and (ii) a speed scaler decides how fast to run.

In this thesis we introduce a preemptive single machine scheduling problem where the machine speed is externally given and depends on the number of jobs that is available for processing. A job is available for processing when it is released but not yet completed. The objective is to minimize the sum of weighted completion times. We will look at some variations of this problem by adding release dates or restricting ourselves to unit weights, unit processing times or by looking at the non-preemptive case.

When the machine speed is constant over time, it is well known that the Smith's rule yields an optimal schedule for both the preemptive and non-preemptive case. Unfortunately this rule gives arbitrary bad results for the problem under consideration.

We introduce a greedy algorithm that solves the problem to optimality when all weights are equal. With only small changes we can alter this algorithm to work when weights are arbitrary and we have unit processing times.

For arbitrary weights and processing times our algorithm finds an optimal schedule when we restrict ourselves to a certain order of job completions. However, we do not know which is an optimal order of job completions. The WSPT-order, which is optimal when machine speed is fixed, can even give arbitrary bad results.

Contents

Chapter 1

Introduction

Energy usage and power management are important issues in recent research. Moore [18] predicted in 1965 that the number of transistors on integrated circuits would double every two years. That prediction came true and the effects of it are now kicking in: the power consumption rate of computing devices has been increasing exponentially. This increased power usage poses two types of difficulties [13]:

- Energy Consumption. As energy is power integrated over time, supplying the required energy may become really expensive, or even technologically infeasible. This is a characteristic difficulty in devices that rely heavily on batteries for energy, and will become even more problematic as battery capacities are increasing at a much slower rate that the power consumption.
- Temperature. The energy used in computing devices is in large part converted to heat. For high-performance processors, cooling solutions are rising at \$1 to \$3 per watt of heat dissipated, meaning that the cooling costs are rising and threaten the computer industry's ability to deploy new systems. Computer processor designers are about to hit a thermic wall.

Therefore computer systems must make a fundamental trade-off between performance and energy usage. The days of 'faster is better' are gone, and energy usage can no longer be ignored in designs. This has led to speed scaling, a technique that adapts the speed of the system to balance energy and performance. Running a job slower saves energy, yet it takes longer and therefore may affect performance. The first paper about speed scaling was written by Yao, Demers and Shenker in 1995 [26]. Their problem was defined as follows:

Problem 1 (Speed scaling)**.** Let $\mathcal{J}$ be a set of jobs and each job $j \in \mathcal{J}$ is associated with a release date r_j, a deadline d_j and a volume p_j. We assume release dates and deadlines are given. We have a variable-speed processor, which is associated with a power function $P(s) = s^\alpha$ with $\alpha \geq 2$. Assuming job j is always processed at a speed of s_j, then j requires v_j/s_j time to be completed. The energy consumption of the processor is integrated over time. A schedule is said to be feasible if the volume of each job is completely processed not before its release date, but before it's deadline. The goal is to find a feasible solution that minimizes the energy consumption.

In this problem minimizing the energy consumption while obeying the deadlines is the only objective. But when there are no deadlines, one can think of a secondary objective as well: minimizing the flow time or sum of completion times. The intuition is that users are willing to pay a certain amount, say x units, of energy to reduce one unit of flow time. Thus a bi-objective would be to optimize total flow time plus x times the amount of energy. Albers and Fujiwara [2] were the first ones to research this bi-objective. A third objective is, given some energy budget, to minimize the sum of completion times or flow time.

Wierman, Andrew and Lin [25] mention three types of speed scaling:

- Dynamic speed scaling: adapting the speed at all times to the current state. A lot of research has been done about dynamic speed scaling, for example by Grunwald, Levis and Morrey [11].
- Static speed scaling: running at a static speed chosen a priori to balance energy and performance.
- Gated-static speed scaling (power-down mechanisms), running at a static speed, except when the machine is idle. We encounter this at an everyday basis: the display of our desktop turns off after some period of inactivity and our laptop transitions to a standby mode if it has been idle for a while.

There already has been a lot of research on speed scaling, particularly on dynamic speed scaling, and there are some interesting results. Some of those results are discussed in Chapter 3. An overview of speed scaling results can be found in the review article by Albers [1].

Fundamentally, when implementing speed scaling, an algorithm must make two decisions at each time: (i) a scheduling policy decides which job(s) to serve, and (ii) a speed scaler decides how fast to run. In this thesis we focus on scheduling policies that, given some speed function such that the machine speed depends on the number of available jobs (active job count), create a schedule that minimizes the sum of completion times.

This problem occurs in our daily life as well. When people have a lot of things to do they tend to work faster than they would when only one or few jobs were assigned to them. Or, when a machine has a lot of jobs to do it needs some of it's capacity to 'remember' these jobs. As soon as the job are completed, they don't have to be remembered anymore and more capacity can be used to process other jobs.

The same problem was proposed on the *Dagstuhl Seminar* in March 2013 by Urtzi Ayestra [3]:

Problem 2 (Problem proposed on Dagstuhl seminar)**.** *'Classical results in size-based scheduling in a single-server queue show that giving preference to short flows is optimal in a wide variety of settings. However all these results typically assume that the speed of the server is constant over time and indeendent of the state of the queue. In this short talk we will show that when the capacity of the system is time-varying (either as a function of the state or as an exogenous process) giving preference to short flows is no longer necessarily optimal, which opens several interesting questions'.*

1.1 Problem Description

In Problem 2 it is said that in single server queues (and single machine scheduling instances) it is often optimal to process small jobs first when the objective is to minimize the flow-time or sum of completion times. In Section 3.1 we show that algorithms as SPT, WSPT and SRPT, all algorithms that prefer small jobs, are indeed optimal for several single machine scheduling problems. In all those problems the machine speed is assumed to be constant. In the light of power-saving it is interesting to vary this speed, and let the machine speed depend on the number of available jobs. In this case it's not always optimal to schedule small jobs first.

We look at the following problem:

Problem 3 (JDMS)**.** Suppose we have n jobs, one machine and a speed function $\overline{s} : \{1, \dots, n\} \to \mathbb{R}$. The machine runs at speed $\overline{s}(i)$, where i is the number of available jobs. Furthermore preemption is allowed. How do we schedule the jobs to minimize the weighted sum of completion times?

We add processing characteristics as release dates, and restrictions as unit processing times and unit weights to this main problem. These additions are put between brackets behind the main problem. For example, when we want to look at JDMS where jobs have release dates, we denote this problem as JDMS(r_j).

We define the machine speed when there are i available jobs as $\overline{s}(i)$, or short: $\overline{s}_i$. When we don't have release dates it sometimes is more convenient to determine the machine speed by the number of jobs that are completed. Therefore we introduce s_i, the speed when i jobs are completed, and thus $n + 1 - i$ jobs are still available.

Chapter 2

Framework and Notation

In this chapter we give a short introduction to scheduling and introduce the notation that will be used in the rest of the thesis. This will only be an introduction in scheduling, and some basic knowledge of complexity theory is presumed. Additional information about scheduling can, amongst others, be found in: *Scheduling: Theory, Algorithms, and Systems* written by Pinedo [20]. For an introduction in complexity theory we refer to *Computational Complexity* by Papadimitriou [19].

In scheduling instances we work with jobs and machines, and we want to schedule the jobs on the machines such that some objective is minimized. In practice we use scheduling to solve a wide variety of problems: to solve scheduling problems in companies that use machines, to make schedules for hospitals and schools, to make the train schedule, etc. The number of jobs used in a schedule is usually expressed as n and the number of machines is expressed as m. A job j has, independent of the machine on which it is processed, a processing time p_j. When job j is processed on machine i this takes time $p_{i,j} = p_j/v_{i,j}$, where $v_{i,j}$ is the speed at which job j is processed on machine i.

Schedules are often visualized in Gantt-charts [8]. In these charts we see at which time a certain job is scheduled on which machine. An example of a Gantt-chart with four jobs and two machines can be found in Figure 2.1. For a better understanding of schedules we use Gantt-charts as visual aid.

2.1 Notation for scheduling problems

A convenient notation for theoretic scheduling problems is introduced by Graham, Lawler, Lenstra and Rinnooy Kan [10]. It consists of three fields: α, β and γ. In the $\alpha|\beta|\gamma$ - structure we organize the properties of a problem. We use α to denote the machine environment, β to describe the processing characteristics and constraints and γ to describe

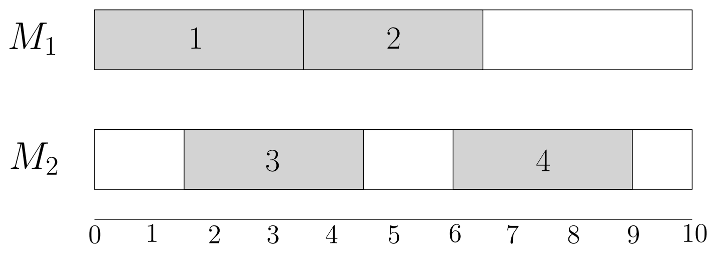

Figure 2.1: Example Gantt-chart

the objective value.

In the *machine environment*, α, we denote with how many and what kind of machines we work. The following environments are commonly used:

1. **Single machine** (1). The case of a single machine is the simplest of all possible machine environments and is a special case of all environments mentioned below. In our problem we only work with the single machine environment.

2. **Parallel machines** (P). We work with identical machines in parallel. In this case $p_{i_1,j} = p_{i_2,j}$ for all machines i_1, i_2.

3. **Related parallel machines** (Q). There are parallel machines with different speeds. The speed of machine i is denoted by v_i. If job j is fully processed on machine i, then this takes time $p_{i,j} = \frac{p_j}{v_i}$. If all machines have the same speed, then this environment is identical to the previous one.

4. **Unrelated machines** (R). This environment is a further generalization of the previous one. There are different machines in parallel, and machine i can process job j with speed $v_{i,j}$. If job j is fully processed on machine i, this will take time $p_{i,j} = \frac{p_j}{v_{i,j}}$. If the speeds of the machines are independent of the jobs, this environment is identical to the previous one.

Several other machine environments are mentioned in the literature, among which: flow shop (F), job shop (J) and open shop (O) and variants of these. These environments are not discussed nor explained here, because we don't need them for our problem.

In the β-field we denote the *processing characteristics and constraints* of the instances. We explain the ones we use in this report below. There are a lot more possible entries for this field though.

1. **Release date** (r_j). If this symbol appears in the β-field it means job j cannot be processed before time r_j. If this symbol is not in the β-field, it means all jobs can start at any time.

2. **Deadline** (d_j). This symbol may appear in the β field, but can be implied by the objective function as well. There are two types of deadlines: 1) strict deadlines, where a schedule is only feasible if all jobs are completed before their deadline. 2) Deadlines that may be exceeded, but there is some penalty when this happens.
3. **Preemptions** (*prmpt*). When *prmpt* is in the β-field, we can interrupt a job at any point in time and put a different job on the machine instead. The amount of processing a preempted job has already received is not lost. When a preempted job is put back on the machine afterwards (this doesn't have to be the same machine as before if there are more machines available), it only needs this machine for the remaining processing time. If *prmpt* is not in the β-field, we have to keep a job on a machine, once started, until its completion.
4. **Unit processing time** ($p_j = 1$). When this appears in the β-field all processing times are equal to 1.
5. **Unit weights** ($w_j = 1$). When this appears in the β-field all weights are equal to 1.

In the γ-field we denote the *objective function.* In this notation the objective is always to minimize some function that depends on the completion times of the jobs. The time job j exists in the system (that is, its completion time on the last machine on which it requires processing) is denoted by C_j.

Examples of possible objective functions are:

1. **Makespan** (C_{max}). This function is defined as $\max(C_1, \ldots C_n)$. It is equal to the completion time of the last job that leaves the system.
2. **Total completion time** ($\sum C_j$). The sum of completion times of n jobs gives an indication of the total holding or inventory costs incurred by the schedule.
3. **Total weighted completion time** ($\sum w_j C_j$). When some jobs have a higher priority than others, or different jobs have a different (inventory)cost it may help to add a weight to the completion times of the jobs.

2.2 Definitions

We also give some definitions that can be helpful to examine properties of problems.

Definition 2.1 (Non-delay schedule)**.** *A feasible schedule is called non-delay if no machine is kept idle while an operation is waiting for processing*

Requiring a schedule to be non-delay is equivalent to prohibiting *unforced idleness.* For many models there are optimal schedules that are non-delay. However, there are models where it may be advantageous to have periods of unforced idleness (Example 2.2).

Example 2.2 (Schedule that requires unforced idleness)**.** *Suppose we have an instance of* $1|r_j|\sum_{j\in J} C_j$*, where* $r_1 = 1, r_2 = 0, p_1 = 2, p_2 = 8$*. Then* σ *in Figure 7.2 is the unique non-delay schedule, while* σ^* *is, according to this objective function, a better schedule.*

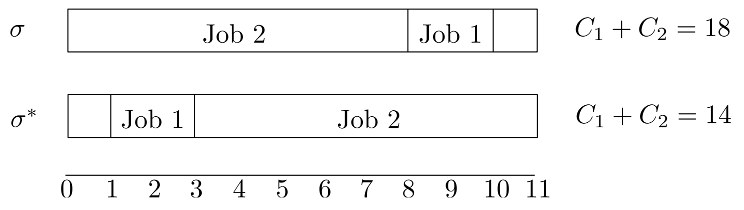

Figure 2.2: Non-delay schedule σ and optimal schedule σ^*.

Definition 2.3 (Available job)**.** *A job is available on time* t *when it is released but not yet completed.*

Definition 2.4 (Clairvoyant)**.** *An algorithm is clairvoyant when all processing times are known when we start processing the job. When the processing time of a job is only known when the job is completed, we speak of a non-clairvoyant model.*

Definition 2.5 (Offline/Online scheduling)**.** *In an offline algorithm all data is known in advance. For an online algorithm we only know the data of the jobs that are released before the current time, and extra jobs can come up any time.*

Even if an algorithm does not always find an optimal solution to the problem, it can be useful to study it. For example when they strongly reduce the time to find a solution that is not so bad compared to the optimum. Furthermore, some algorithms give a lot of insight in the problem, though they do not find an optimal solution.

Definition 2.6 (c-approximation algorithm)**.** *An algorithm is a* c*-approximation algorithm when the objective value of the solution given by the algorithm is at most* c *times the objective value of the optimal solution.*

Chapter 3

Related Work

When implementing speed scaling, an algorithm must make two decisions at each point in time: (i) a scheduling policy decides which job(s) to serve, and (ii) a speed scaler decides how fast to run. We already know there are three types of speed scaling: static, static-gated and dynamic. Instead of making both decisions at the same time one can also first determine some speed function. This speed function depends for example on time or the number of available jobs and make a schedule that is optimal according to that predetermined speed function. JDMS is about scheduling problems with an arbitrary externally given speed function that depends on the number of available jobs.

In the first section we have a look at results for the well-known single machine problems where the machine speed is constant. In Section 2 we consider some problems that relate to JDMS, but are slightly different.

3.1 Single machine problems with fixed machine speed

In this section we are going to sum up some of the well-known results about single machines that have a fixed machine speed. These results, especially the way of proving theorems, are useful for JDMS as we mimic some of the methods to obtain results for JDMS. If the problems are in *P*, we will give a polynomial algorithm that solves the problem. If the problem is in *NP* we will prove that by giving a reduction from a *NP*-complete or *NP*-hard problem. The problems we will have a look at can be found in Figure 3.1. An overview of complexity results for single-machine problems can be found in [5].

Note that the problem $1|prmpt, r_j, p_j = 1|\sum w_j C_j$ is an open problem.

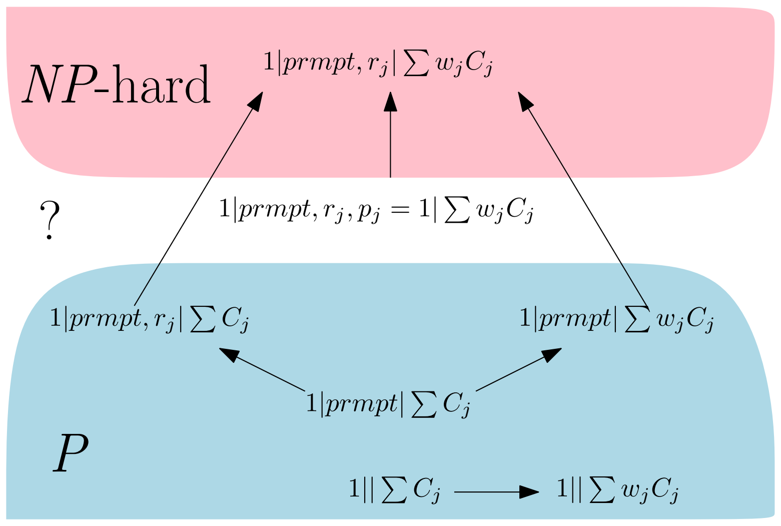

Figure 3.1: Overview of single machine problems.

3.1.1 The total weighted completion time

In the $\alpha|\beta|\gamma$ structure we denote this problem as $1||\sum w_jC_j$. The weight of a job j may be regarded as an importance factor. For example: it may represent a holding cost per unit time. This problem gives rise to one of the best known rules in scheduling theory, the so-called *Weighted Shortest Processing Time first (WSPT)* rule or Smith's rule [22]. According to this rule jobs are processed in decreasing order of w_j/p_j.

Theorem 3.1. *The WSPT rule is optimal for* $1||\sum w_jC_j$ *[22].*

Proof. We prove this theorem by contradiction. Suppose a schedule σ, that differs from the WSPT schedule, is optimal. In this schedule there must be at least two adjacent jobs, say job j followed by job k, such that:

$$\frac{w_j}{p_j} < \frac{w_k}{p_k}.$$

Perform a so-called *Pairwise Interchange* on jobs j and k and call the new schedule σ'. While under the original schedule σ job j starts processing at time t and is followed by job k, under the new schedule σ' job k starts processing at time t and is followed by job j. All other jobs remain in their original position. The completion times of the jobs processed before jobs j and k are not affected by the interchange. Neither are the completion times of the jobs processed after jobs j and k. Thus the difference in the objective values under schedules σ and σ' is due to only jobs j and k (see Figure 3.2).

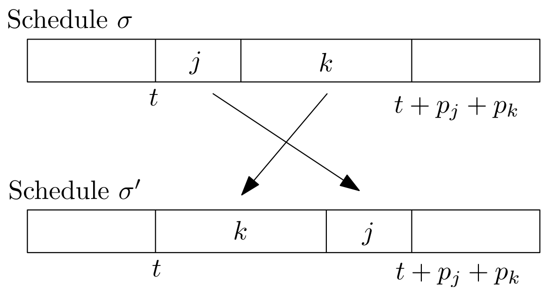

Figure 3.2: A pairwise interchange of jobs j and k.

Under σ the weighted completion time of jobs j and k is

$$(t + p_j)w_j + (t + p_j + p_k)w_k, \tag{3.1}$$

while under σ' it is

$$(t + p_k)w_k + (t + p_k + p_j)w_j. \tag{3.2}$$

We subtract (3.2) from (3.1) to obtain equation (3.3).

$$(t + p_j)w_j + (t + p_j + p_k)w_k - (t + p_k)w_k - (t + p_k + p_j)w_j \tag{3.3}$$

When we simplify this equation, as the majority of the terms sum up to zero. We obtain equation (3.4)

$$p_j w_k - p_k w_j \tag{3.4}$$

As $w_j/p_j < w_k/p_k$, the sum of the two weighted completion times under σ' is strictly less than under σ. This contradicts the optimality of σ and completes the proof. $\square$

The *Shortest Processing Time (SPT)* rule processes the jobs such that the processing time in this sequence are non-decreasing. The problem $1||\sum C_j$ is a special case of $1||\sum w_j C_j$, as it happens to be the case where $w_j = 1$ for all j. According to Theorem 3.1 we have to schedule the jobs in non-increasing order $\frac{1}{p_j}$. In other words, in that case we schedule the jobs in non-decreasing order p_j.

Corollary 3.2. *SPT is optimal for* $1||\sum C_j$.

Problem $1||\sum w_j C_j$ has a geometric interpretation as well, using 2D-Gantt charts. This interpretation was introduced by Eastman, Even and Isaacs [6]. According to Theorem 3.1 processing the jobs in this order wil lead to an optimal solution with value $\sum_{1 \leq i \leq j \leq n} p_i w_j$. This sum equals the area of the rectangles in Figure 3.3.

The total size of the rectangles is minimized when the jobs are numbered in non-decreasing order w_i/p_i.

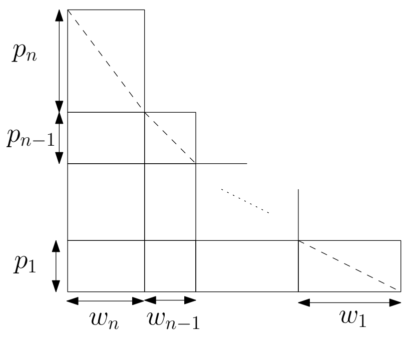

Figure 3.3: 2D-Gantt chart

3.1.2 The total weighted completion time with preemptions

Suppose we have a single machine and it is allowed to preempt jobs. The objective is to minimize the weighted sum of completion times. Then we prove that preemption will not help to improve the schedule.

Lemma 3.3. *An optimal schedule for* $1|prmpt|\sum w_j C_j$ *does not use preemption.*

Proof. Suppose there is an optimal schedule σ that uses preemption. Let job i be the first job that is preempted by some job j. Then we change σ to σ' by interchanging some operations.

If the case is $C_i \leq C_j$:

1. Take the first p_i units of time that were devoted to either job i and j after time t, and use them instead to process job i to completion.
2. Take the remaining p_j units of time that were spend processing job i and j after time t and use them to schedule job j.

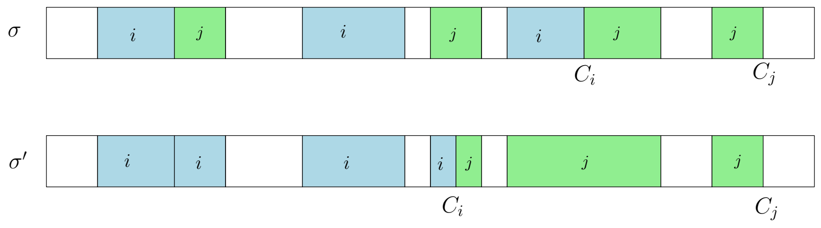

Figure 3.4: Visualized interchange argument.

If $C_i > C_j$ we take the first p_j units of time that were devoted to either job i and j after time t, and use them instead to process job j to completion. Then take the remaining p_j

units of time that were spend processing job i and j after time t and use them to schedule job i.

We compare the objective value of σ and σ'. In both cases $\max(C_j^{\sigma'}, C_i^{\sigma'}) = \max(C_j^{\sigma}, C_i^{\sigma})$ and $\min(C_j^{\sigma'}, C_i^{\sigma'}) < \min(C_j^{\sigma}, C_i^{\sigma})$. In both cases the other completion times stay the same and therefore in both cases it holds that $\sum w_j C_j^{\sigma} > \sum w_j C_j^{\sigma'}$. Thus σ' is a better schedule than σ, contradicting the fact that σ is optimal. As the assumption that there exists an optimal schedule that uses preemption leads to a contradiction, there is no optimal schedule that uses preemption. □

Corollary 3.4. *WSPT is optimal for* $1|prmpt|\sum w_j C_j$.

Proof. As an optimal schedule for $1|prmpt|\sum w_j C_j$ does not use preemption, it has the same optimal value as the optimal schedule for $1||\sum w_j C_j$. Therefore we conclude from Theorem 3.1 that $WSPT$ is optimal for $1|prmpt|\sum C_j$. □

Corollary 3.5. *SPT is optimal for* $1|prmpt|\sum C_j$.

This follows from Corollary 3.4. The problem $1|prmpt|\sum C_j$ is a special case of problem $1|prmpt|\sum w_j C_j$: the case where $w_j = 1$ for all j. According to Corollary 3.4 we have to schedule the jobs in non-increasing order $\frac{1}{p_j}$. In other words, we have to schedule the jobs in non-decreasing order p_j, which is exactly the order of SPT.

3.1.3 The total completion time with release dates and preemptions.

In the previous subsections we assumed all jobs are immediately available. In practice this is not always the case and therefore we will have a look at the problem where jobs have release dates. The release date of a job is the time that it becomes available, jobs cannot be processed before their release date. This problem is denoted as $1|prmpt, r_j|\sum C_j$.

Every job i has a remaining processing time p_i', this is the time that a job still needs to be completed. The algorithm *Shortest Remaining Processing Time* (SRPT) processes the job with the shortest remaining processing time and is available at that moment.

L. Schrage has proven that SRPT is optimal for this problem.

Theorem 3.6. *SRPT is optimal for* $1|prmpt, r_j|\sum C_j$. *[21]*

Proof. Consider an optimal schedule σ in which available job i with the shortest remaining processing time is not being processed at time t, and instead available job k is being processed. Let p_i' and p_k' denote the remaining processing times for jobs i and k at time t, so $p_i' < p_k'$. In total $p_i' + p_k'$ time units are spent on jobs i and k after time t. We look at all the time intervals after time t that process either job i or k to obtain another schedule σ':

1. Take the first p'_i units of time that were devoted to either of jobs i and k after time t, and use them instead to process job i to completion.

2. Take the remaining p'_k units of time that were spent processing jobs i and k after time t, and use them to schedule job k.

As $p'_i < p'_k$, we know that $C_i^{\sigma'} < \min(C_k^\sigma, C_i^\sigma)$. In both σ and σ' the same time intervals are used to schedule job i and k, thus $C_k^{\sigma'} = \max(C_k^\sigma, C_i^\sigma)$. All other completion times remain the same, thus the objective value of σ' wil be smaller than the objective value of σ. This contradicts the optimality of σ.

Thus SRPT is optimal for $1|prmpt|\sum C_j$. □

3.1.4 The total weighted completion time with preemptions and release dates

The total weighted completion time with preemptions and release dates is strongly NP-hard. This has been proven by J. Labetoulle, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinooy Kan in [14]. This means we can transform any instance from some strongly NP-hard problem to an instance of $1|prmpt, r_j|\sum w_j C_j$ in polynomial time. As this proof is rather complicated, we start with a proof that says $1|prmpt, r_j|\sum w_j C_j$ is weakly NP-hard by making a reduction from the PARTITION problem.

Problem 4 (PARTITION)**.** Given a set $J = \{1, \ldots, m\}$ and positive integers $a_1, \ldots, a_m$, do there exist two disjoint subsets $S_1, S_2 \subset J$ with $S_1 \cup S_2 = J$ such that $\sum_{j \in S_1} a_j = \sum_{j \in S_2} a_j$?

We define $b = \frac{1}{2}\sum_{j \in J} a_j$. Using PARTITION, we prove the following theorem:

Theorem 3.7. *$1|prmpt, r_j|\sum w_j C_j$ is weakly NP-complete.*

Proof. We prove this theorem by making a reduction from the weakly NP-complete problem PARTITION. The reduction is as follows: we use $m+1$ jobs, that are defined in the tabel below:

m normal jobs				1 dummy job		
r_j	$=$	0	$j \in J$	r_{m+1}	$=$	b
p_j	$=$	a_j	$j \in J$	p_{m+1}	$=$	1
w_j	$=$	a_j	$j \in J$	w_{m+1}	$=$	2

We claim that there there is a solution for PARTITION if and only if the instance of $1|prmpt, r_j|\sum w_j C_j$ has an objective value smaller or equal to U_1, where U_1 is defined as follows:

$$U_1 = \sum_{1 \le j \le k \le m} a_j a_k + 3b + 2 \tag{3.5}$$

We prove that if some instance $\mathcal{I}$ is a yes-instance for PARTITION, then it will be a yes-instance for $1|prmpt, r_j|\sum w_j C_j$, by creating the following schedule:

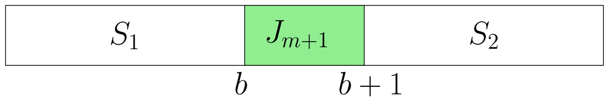

Figure 3.5: Schedule that corresponds to the solution of PARTITION.

Jobs in S_1 and S_2 are not preempted and processed in arbitrary order. When the dummy job is left out, we would have had total weighted completion time $\sum_{1\leq j\leq k\leq m} a_j a_k$. If the dummy job is scheduled at b, between S_1 and S_2, it will increase the objective value in two ways: (i) by its own weight and completion time, it will add $2(b+1)$ to the objective value and (ii) the objective value will increase by b as S_2 is shifted one time unit to the right. Thus this schedule will have a total objective value of $\sum_{1\leq j\leq k\leq m} a_j a_k + 3b + 2 = U_1$.

We prove that the schedule in Figure 3.5 is the only schedule with a total weighted completion time smaller or equal than U_1. We first prove that we only have to look at schedules where the dummy job is not preempted. Suppose dummy job $m+1$ is preempted. Then we change σ to σ' by putting the first part of $m+1$ before the second part, and shift the jobs that were in between to the left (and leave the other jobs unchanged, Figure 3.6). As $C^{\sigma}_{m+1} = C^{\sigma'}_{m+1}$ and the other jobs remain in the same place, or are processed earlier, it holds that $C^{\sigma'}_j \leq C^{\sigma}_j$ for all jobs j. Therefore σ' is at least as good as σ. We repeat this argument until dummy job $m+1$ is no longer preempted.

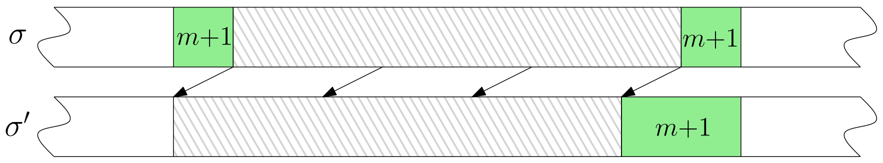

Figure 3.6: Changing σ to σ'.

If we only schedule jobs $1, \ldots, m$ we have to solve a scheduling problem without release dates. Theorem 3.4 tells us that WSPT will result in an optimal solution. As $w_j/p_j = a_j/a_j = 1$ for all jobs $1, \ldots, m$ any nonpreemptive schedule without machine idle time will be optimal and has value $\sum_{1\leq j\leq k\leq m} a_j a_k$. Inserting the dummy job in the schedule increases the objective value by the total weight of all jobs completed after that dummy job.

We first define some variables that we use in our proof. Let us denote the index set of all normal jobs completed before J_{m+1} by X_1. We define the length of the interval from b (the release date of J_{m+1}) until J_{m+1} starts as c. Thus $c \geq 0$ and when $c = 0$, dummy job $m+1$ starts at its release date. We define the length of the interval from the end of X_1 until the start of J_{m+1} by d. Note that $d \geq 0$ and when $d = 0$, all jobs that are started before J_{m+1} will also be completed before dummy job $m+i$ (Figure 3.7).

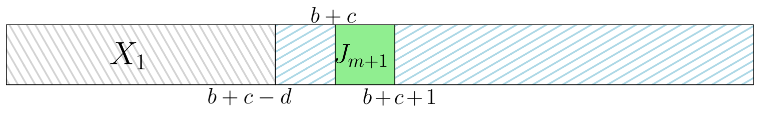

Figure 3.7: Variables c and d visualised.

We have for any schedule that:

$$C_{m+1} = b + c + 1 \tag{3.6}$$

As $w_i = p_i$ for all normal jobs $1 \leq i \leq m$, it holds that:

$$\sum_{i \in X_1} w_i = \sum_{i \in X_i} p_i = b + c - d. \tag{3.7}$$

By inserting the dummy jobs in the original schedule we increase the original value by $2C_{m+1} + 2b - \sum_{i \in X_1} w_i$: $2C_{m+1}$ for the inserted dummy job and $2b - \sum_{i \in X_1} w_i$ as we shift the latter part of the schedule to the right by 1 when we insert the dummy job.

Therefore, the objective value of any schedule is:

$$\sum_{1 \leq j \leq k \leq m} a_j a_k + 2C_{m+1} + \left(2b - \sum_{i \in X_1} w_i\right) \tag{3.8}$$

We combine 3.6, 3.7 and 3.8 to obtain the next equation:

$$\sum_{1 \leq j \leq k \leq m} a_j a_k + 2(b + c + 1) + 2b - (b + c - d)$$

We simplify the previous equation:

$$\sum_{1 \leq j \leq k \leq m} a_j a_k + 3b + 2 + c + d$$

We substitute part of the equation by (3.5)

$$U_1 + c + d \tag{3.9}$$

Thus the objective value of any schedule can be written as (3.9).

Suppose we have found a yes-instance $\mathcal{I}$ for $1|prmpt, r_j|\sum w_j C_j$, then the objective value of the schedule is less or equal to U_1. As $c \geq 0$ and $d \geq 0$, it has to hold that $c = d = 0$. Then we have a schedule as described in Figure 3.5. Then X_1 and $J \setminus X_1$ are two pairwise disjoint sets with $\sum_{j \in X_1} p_j = \sum_{j \in J \setminus X_1} p_j$. Thus $\mathcal{I}$ is a yes-instance for PARTITION.

$\square$

We are going to extend the previous result. We make a reduction from 3-PARTITION to proof $1|prmpt, r_j|\sum w_j C_j$ is not only weakly NP-hard, but also strongly NP-hard. This problem is defined as follows:

Problem 5 (3-PARTITION)**.** Given a set $J = \{1, \dots, 3m\}$ and positive integers $a_1, \dots, a_{3m}$ with $\frac{1}{m}\sum_{j\in J} a_j = b$ and $\frac{1}{4}b < a_j < \frac{1}{2}b$ $(j \in J)$, does there exist m pairwise disjoint subsets $S_i \subset J$ such that $\sum_{j\in S_i} a_j = b$ for $i = 1, \dots, m$?

Using 3-PARTITION, we prove the following theorem:

Theorem 3.8. *The problem $1|prmpt, r_j|\sum w_j C_j$ is NP-hard.*

Proof. We prove Theorem 3.8 by making a reduction from the NP-complete problem 3-PARTITION:

The reductions is as follows: we use $4m - 1$ jobs, that are defined in the table below.

$3m$ normal jobs				$m-1$ dummy jobs			
r_j	$=$	0	$j \in J$	r_j	$=$	$(j-3m)(b+1)-1$	$j = 3m+1, \dots 4m-1$
p_j	$=$	a_j	$j \in J$	p_j	$=$	1	$j = 3m+1, \dots 4m-1$
w_j	$=$	a_j	$j \in J$	w_j	$=$	2	$j = 3m+1, \dots 4m-1$

Note that this reduction is similar to the one used to proof Theorem 3.7. We claim that there exist a 3-partition if and only if the instance of $1|prmpt, r_j|\sum w_j C_j$ has an objective values smaller or equal to U_2, where U_2 is defined as follows:

$$U_2 = \sum_{1\le j\le k\le 3m} a_j a_k + (m-1)m(\frac{3}{2}b+1). \tag{3.10}$$

When $m = 2$, we have an instance of PARTITION and in this case $U_1 = U_2$. We show that when there is a 3-partition with m pairwise disjoint sets $S_1, \dots S_m$, then the corresponding schedule for $1|prmpt, r_j|\sum w_j C_j$ where the objective value is at most U_2, is as follows:

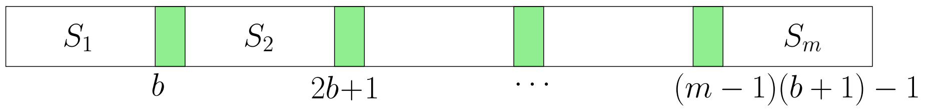

Figure 3.8: Schedule that corresponds to the solution of 3-partition

The total weighted completion time of the schedule in Figure 3.8 can be computed as follows:

If $S_1, \dots S_m$ are all scheduled without dummy jobs in between, the objective value would be $\sum_{1\le j\le k\le 3m} a_j a_k$

When dummy job $3m+i$ is scheduled after S_i, it increases the objective value by $2C_{3m+1}+(m-i)b$. As $C_{3m+1}=i(b+1)$, the total objective value is increased by $2i(b+1)+(m-i)b = ib+2i+mb$

When we add $m-1$ dummy jobs to the $3m$ normal jobs, we obtain objective value

$$\sum_{1\leq j\leq k\leq 3m} a_j a_k + \sum_{i=1}^{m-1} (ib+2i+mb)\,.$$

This can be simplified to:

$$\sum_{1\leq j\leq k\leq 3m} a_j a_k + \frac{1}{2}m(m-1)b + m(m-1) + (m-1)mb.$$

Simplify even further to obtain the next equation:

$$\sum_{1\leq j\leq k\leq 3m} a_j a_k + (m-1)m(\frac{3}{2}b+1).$$

This is exactly U_2. Thus the schedule in Figure 3.8 has objective value U_2.

We prove that the schedule in Figure 3.8 is the only schedule that has a total weighted completion time less or equal to U_2. To obtain this result we first prove that we only have to look at schedules where the dummy jobs are not preempted. Suppose we have some optimal schedule σ, then we look at all the time units where dummy jobs are processed. As all dummy jobs have equal weight and processing time, we assume that dummy jobs are scheduled in order of their release dates, without preempting each other.

So when we only look at the time units where dummy jobs are processed, they won't be preempted. Though, looking at the whole schedule, dummy jobs can still be preempted by the first $3m$ normal jobs. Suppose job $3m+i$ is the first dummy job that is preempted. Then we change σ to σ' by putting the first part of $3m+i$ before the second part, and shift the jobs that were in between to the left and leave the other jobs unchanged (Figure 3.9). As $C_i^{\sigma} = C_i^{\sigma'}$ and the other job remain in the same place, or are processed earlier, it holds that $C_j^{\sigma} = C_j^{\sigma'}$ for all jobs j. Therefore σ' is at least as good as σ. We repeat this argument until all dummy jobs are schedule without being preempted.

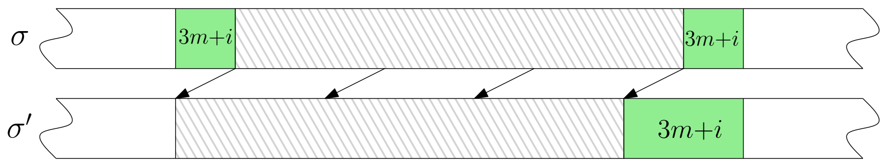

Figure 3.9: Changing σ to σ'.

We give the objective value for all schedules that potentially could have an objective value smaller or equal to U_2. If we only schedule jobs $1,\ldots,3m$ we have to solve a scheduling problem without release dates. Theorem 3.4 tells us that WSPR results in an optimal solution. As $w_j/p_j = a_j/a_j = 1$ for all jobs $1, \dots, 3m$ any nonpreemptive schedule without machine idle time will be optimal and has value $\sum_{1 \leq j \leq k \leq 3m} a_j a_k$. Inserting a unit-dummy job in a schedule increases the contribution to the objective value of the latter set by the total weight of all jobs completed after that dummy job. By repeating this process, inserting the $i'th$ dummy job after the previous dummy job is completed, we insert all dummy jobs in the original schedule and determine the increase in the objective value (note that we have proven that is is not useful to preempt dummy jobs).

We first define some variables that we use in our proof. Let us denote the index set of all normal jobs completed before J_{3m+i} by X_i, the length of the interval from $(b+1)i - 1$ (its release date) until J_{3m+i} starts by c_i, and the length of the interval from the last completion before J_{3m+i} until the start of J_{3m+i} by d_i. Note that when $d_i = 0$, this means all jobs that start before dummy job $3m+i$ will also be completed before dummy job i. Clearly $d_i \geq 0$ and $c_i \geq 0$ (Figure 3.10).

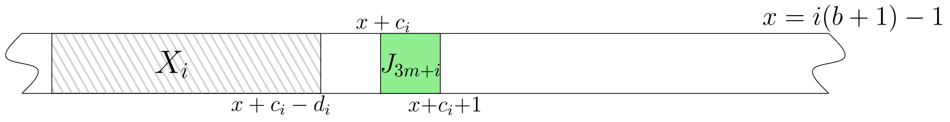

Figure 3.10: Variables c_i and d_i visualised.

Then we have for any schedule that:

$$C_{3m+i} = i(b+1) + c_i, \qquad \text{for all dummy jobs } 3m+i \tag{3.11}$$

As $w_i = p_i$ for all normal jobs $1 \leq i \leq 3m$, it holds that:

$$\sum_{j \in X_i} w_j = \sum_{j \in X_i} p_j = ib + c_i - d_i. \tag{3.12}$$

By inserting $m-1$ dummy jobs in the original schedule we increase the original value by $2\sum_{i=1}^{m-1} C_i$ (the weight and completion time from the dummy jobs) and as we shift the latter part of the schedule by 1 for every dummy job we insert, we also get an increase of $\sum_{i=1}^{m-1} \left(mb - \sum_{j \in X_i} w_j \right)$.

Therefore, the objective value of any schedule will be:

$$\sum_{1 \leq j \leq k \leq 3m} a_j a_k + 2 \sum_{i=1}^{m-1} C_{3m+i} + \sum_{i=1}^{m-1} \left(mb - \sum_{j \in X_i} w_j \right) \tag{3.13}$$

We combine 3.11, 3.12 and 3.13 to obtain the next equation:

$$\sum_{1 \leq j \leq k \leq 3m} a_j a_k + 2 \sum_{i=1}^{m-1} (i(b+1) + c_i) + \sum_{i=1}^{m-1} (mb - (ib + c_i - d_i)) \tag{3.14}$$

We rewrite the latter part of the equation as one sum.

$$\sum_{1\leq j\leq k\leq 3m} a_j a_k + \sum_{i=1}^{m-1} (2(ib + i + c_i) + (mb - ib - c_i + d_i)) \tag{3.15}$$

We simplify the previous equation:

$$\sum_{1\leq j\leq k\leq 3m} a_j a_k + \sum_{i=1}^{m-1} (2i + ib + mb + c_i + d_i) \tag{3.16}$$

Splitting the sum:

$$\sum_{1\leq j\leq k\leq 3m} a_j a_k + \sum_{i=1}^{m-1} 2i + \sum_{i=1}^{m-1} ib + \sum_{i=1}^{m-1} mb + \sum_{i=1}^{m-1} c_i + \sum_{i=1}^{m-1} d_i \tag{3.17}$$

Computing the first three sums:

$$\sum_{1\leq j\leq k\leq 3m} a_j a_k + m(m-1) + \frac{1}{2}m(m-1)b + (m-1)mb + \sum_{i=1}^{m-1} c_i + \sum_{i=1}^{m-1} d_i \tag{3.18}$$

We simplify again:

$$\sum_{1\leq j\leq k\leq 3m} a_j a_k + m(m-1)(\frac{3}{2}b + 1) + \sum_{i=1}^{m-1} c_i + \sum_{i=1}^{m-1} d_i \tag{3.19}$$

We substitute part of the equation by (3.10)

$$U_2 + \sum_{i=1}^{m-1} c_i + \sum_{i=1}^{m-1} d_i \tag{3.20}$$

Thus the objective value of any schedule can be written as (3.20).

Suppose we have found a yes-instance $\mathcal{I}$ for $1|prmpt, r_j|\sum w_j C_j$, then the objective value of the schedule is less or equal to U_2. As $c_i \geq 0$ and $d_i \geq 0$ for all $1 \leq i \leq m-1$, it has to hold that $c_i = d_i = 0$. Therefore there must be pairwise disjoint sets $X_1, \dots X_{m-1}, J \setminus \{X_1 \cup \dots \cup X_{m-1}\}$ such that $\sum_{j\in S_i} p_j = b$. Thus $\mathcal{I}$ is a yes-instance for 3-partition. $\square$

3.2 Related Problems

In this section, we discuss three problems that are related to JDMS. One of the most extensively studied speed scaling problems was the problem in a paper by Yao, Demers and Shenkers [26]:

Problem 6 (Speed scaling)**.** Let $\mathcal{J}$ be a set of jobs and each job $j \in \mathcal{J}$ is associated with a release time r_j, a deadline d_j and a volume p_j. We assume release dates and deadlines are given integers. We have a variable-speed processor, which is associated with a power function $P(s) = s^\alpha$ with $\alpha \geq 2$. The energy consumption of the processor is the power function integrated over time. Assuming job j is always processed at a speed of s_j, then j requires v_j/s_j time to be completed. A schedule is said to be feasible if the volume of each job is completely processed not before its release date, but for its deadline. The goal is to find a feasible solution that minimizes the energy consumption.

The second problem is discussed by Gawiejnowicz in [9]. His problem is formulated as follows:

Problem 7 (Gawiejnowicz)**.** Suppose we work with a single processor, where the speed of the processor depends on the number of completed jobs and is described by a externally given function. After k jobs there has to be a break, and after that the speed varies again. How do we schedule the jobs to minimize the makespan?

Problem 7 differs from JDMS on several points. The most important difference is the objective function of the problem: Gawiejnowicz wants to minimize the makespan, while we try to minimize the sum of weighted completion times. Another difference is that preemption not allowed, while we do allow preemption in JDMS. Finally, Gawiejnowicz introduces breaks in the schedule: after k jobs are processed there has to be some break. If $k \geq n$ no break has to be scheduled.

We will also have a look at the following problem:

Problem 8 (Time dependent machine speed)**.** Suppose we work with a single processor, where the speed of the processor depends on the time. Jobs have release dates, weights and processing times. How do we schedule the jobs to minimize the sum of weighted completion time?

This problem differs from JDMS as the speed function does not depend on the number of available jobs, but on the time instead. Research on this subject has been extensive (among others: [7], [24], [17]) and there are some interesting recent discoveries.

3.2.1 Fundemental algorithms for speed scaling

One of the most extensively studied speed scaling problems was the problem in a paper by Yao, Demers and Shenkers [26]. The problem by Yao et al. assumes there is no upper bound on the maximum processor speed. Hence there always exists a feasible schedule satisfying all job deadlines. Furthermore, it is assumed that a continuous spectrum of speeds is available. This framework is by far the most extensively studied algorithmic speed scaling problem. There are three fundamental algorithms for speed scaling, all introduced in [26]:

1. *Algorithm YDS.* YDS is referring to the initials of the authors: Yao, Demers and Shenker. The algorithm proceeds in a series of iterations. In each iteration, a time interval of maximum density is identified and a corresponding partial schedule is constructed. Loosely speaking, the density of an interval $\mathcal{I}$ is the minimum average speed necessary to complete all jobs that must be scheduled in $\mathcal{I}$. A high density requires a high speed. Formally, the density $\Delta_{\mathcal{I}}$ of a time interval $\mathcal{I} = [t, t']$ is the total work to be completed in $\mathcal{I}$ divided by the length of $\mathcal{I}$. More precisely, let $S_{\mathcal{I}}$ be the set of jobs J_i that must be processed in $\mathcal{I}$ because their release time and deadline are in $\mathcal{I}$, thus: $[r_i, d_i] \in \mathcal{I}$. The corresponding total processing volume is:

   $$\Delta_{\mathcal{I}} = \frac{1}{|\mathcal{I}|} \sum_{J_i \in S_{\mathcal{I}}} p_i$$

   Algorithm YDS repeatedly determines the interval $\mathcal{I}$ of maximum density. In such an interval $\mathcal{I}$, the algorithm schedules the jobs of $S_{\mathcal{I}}$ at speed $\Delta_{\mathcal{I}}$ using the *Earliest Deadline First (EDF)* policy. This well-known policy always executes the job having the earliest deadline amongst the available unfinished jobs. We give a small example of Algorithm YDS:

   Example 3.9 (Algorithm YDS)**.** *Suppose we have an instance with 5 jobs, specified as* $J_i = (r_i, d_i, p_i)$*.* $J_1 = (0, 24, 6); J_2 = (3, 8, 7); J_3 = (5, 7, 4); J_4 = (13, 20, 4); J_5 = (15, 18, 3)$*. Interval* $[3, 8]$ *has the highest density:* $\Delta_{[3,8]} = 2.20$ *and* $S_{[3,8]} = \{J_2, J_3\}$*. Thus we schedule* J_2 *and* J_3 *with speed* 2.20 *in interval* $[3, 8]$*. Excluding interval* $[3, 8]$*, interval* $[13, 20]$ *has highest density:* $\Delta_{[13,20]} = 1$ *and* $S_{[3,8]} = \{J_4, J_5\}$*. Thus we schedule* J_4 *and* J_5 *in interval* $[13, 20]$ *with speed* 1*. Then only* J_1 *is left, with* $\Delta_{[0,3]\cup[8,13]\cup[20,24]} = 0.5$ *and* $S_{[0,3]\cup[8,13]\cup[20,24]} = \{J_1\}$*. This results in the schedule in Figure 3.11.*

   Note that this is an offline-algorithm: all information is known.

2. *Algorithm average rate.* For any arriving job J_i, *Average Rate* considers the density $\delta_i = p_i/(d_i - r_i)$, which is the minimum average speed necessary to complete the job in time if no other jobs were present. At any time t, the speed $s(t)$ is set to the accumulated density of jobs active at time t. A job J_i is active at time t if it is available for processing at that time, thus if $t \in [r_i, d_i]$. Available jobs are scheduled according to the EDF policy. In short: at any time t, use a speed of:

   $$s(t) = \sum_{J_i : t \in [r_i, d_i]} \delta_i.$$

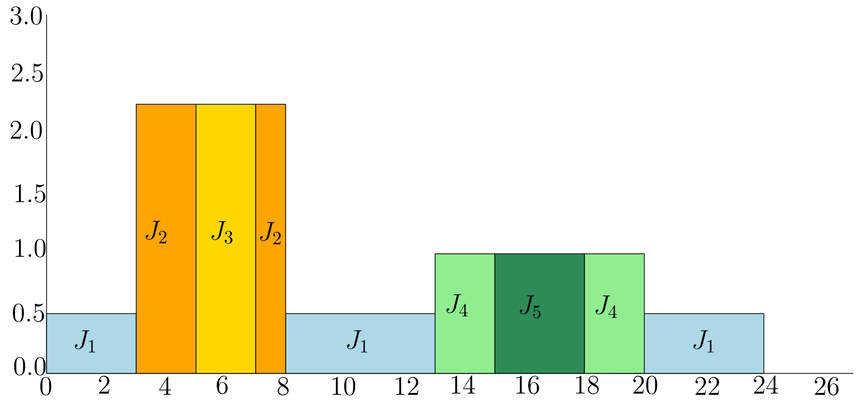

Figure 3.11: Optimal schedule according to Algorithm YDS.

If energy usage is s^α with $\alpha > 2$ for machine speed s, then the approximation guarantee is bounded by $2^{\alpha-1}\alpha^\alpha$

3. *Algorithm optimal rate.* This algorithm computes the optimal schedule for the jobs that are currently known (using YDS). This algorithm has a approximation guarantee of exactly α^α and is therefore better than *Algorithm Average Rate.*

In practice the machine speed is bounded, and often only a finite set of discrete speed levels $s_1 < s_2 < ... < s_d$ is available. With some small changes Algorithm YDS can still find an optimal schedule. We first construct the schedule according to YDS. For each identified interval $\mathcal{I}$ of maximum density, we approximate the desired speed $\Delta_\mathcal{I}$ by the two adjacent speed levels s_k and s_{k+1}, such that $s_k < \Delta_\mathcal{I} < s_{k+1}$. Speed s_{k+1} is used first for some δ time units and s_k is used for the last $|\mathcal{I}| - \delta$ time units in $\mathcal{I}$, where δ is chosen such that the total work completed in $\mathcal{I}$ is equal to the original amount of $|\mathcal{I}|\Delta_\mathcal{I}$.

These were the first results on speed scaling by Yao et al.

3.2.2 Gawiejnowicz: minimizing the makespan

Without loss of generality, the speeds are chosen from the interval $[0, 1]$. This problem matches situations where machines consists of devices with their own renewable power source (for example: batteries) or when we consider the work of human groups (construction workers, painters, etc). Examples of the settings are hard physical work where the execution of a job causes a tiredness of workers such that the next job is processed with a lower speed. Or the other way around, the processing on a machine which gets hot during work up to the point of time when further growth in temperature could cause its destroying. In both cases after some time a break is needed.

In his paper Gawiejnowicz proves that, for an arbitrary speed functions, it is best to process the job with the largest processing time at the position where is it processed at the highest speed, the second largest processing time at the position where it is processed with the second largest speed, etc. This results in an optimal solution.

Let v_j be the speed when j jobs are completed. The algorithm is as follows:

```
begin
  Sort $p_j$ in non-increasing order
  For j:=1 to $n$ do
  {    $s_{1,j} = v_j$,
       $s_{2,j} = j$    }
  Sort $s_{j,1}$ in non-increasing order
  Reindex $s$ according to $s_{j,1}$ values
  For j:=1 to $n$ do
       Insert $p_j$ at the $s'_{j,2}th$ place in the list $L$
end
```

Thus this problem is solvable in polynomial time. We see that JDMS without preemption can be solved in a similar way.

3.2.3 Time dependent machine speed

The problem that is most related to JDMS is Problem 8, where, in contrary to being dependent on the number of available jobs, the machine speeds depends on the moment in time. There are a lot of practical examples where the machine speed indeed depends on time. Think about computers that become slower over the years or faster after some heat-up period, people that work slower because of their age or planned machine reparations that make the machine completely unavailable for some time.

This problem is being discussed for over 30 years already and it is only recently, in 2010, that one found a constant approximation algorithm for minimizing the function $\sum w_j C_j$. This was a $(4+\epsilon)$-approximation by Epstein et al [7]. This guarantee holds for any given speed function. When the speed is only increasing, there is an approximation algorithm with approximation-rate $(\sqrt{3}+1)/2$. This scheme was found by Stiller and Wiese in 2010 [23]. When release dates are added, and we work with arbitrary speed functions, the problem is proven to be NP-hard, even when for each job the weight and processing time are equal [24].

In 2013 Megow and Verschae [17] came up with an efficient PTAS for minimizing the total weighted completion time on a machine of varying speed and improved the old PTAS with rate $(1+\epsilon)$. They use an interesting technique, a variant of 2D-Gantt chart interpretation (described in section 3.1.1). Their PTAS can be found in [17].

Chapter 4

JDMS without preemption

In this chapter we consider the problem JDMS as stated in Chapter 1, with the restriction that jobs cannot be preempted:

Problem 9 (JDMS without preemption)**.** Suppose we have n jobs, one machine and a speed function s. The machine speed depends on the number of available jobs. How do we schedule the jobs to minimize the sum of weighted completion times?

Note that this problem is similar to $1||\sum w_jC_j$, except that the machine speed depends on the available jobs. Therefore $1||\sum w_jC_j$ is a special case of JDMS without preemption.

4.1 JDMS($w_j = 1$) without preemption

In this subsection we show how to construct an optimal schedule for JDMS($w_j = 1$) without preemption. The algorithm is similar to the algorithm by Gawiejnowicz [9], where he looked at the same problem, but tried to minimize the makespan instead. Suppose we have n jobs with processing times $p_1 \leq \cdots \leq p_n$. We know that the job on position i is processed with speed s_i. We denote the processing time and completion time of the job that is processed on position i as $p_{(i)}$ and $C_{(i)}$. We write the completion time of job i as:

$$C_{(i)} = \sum_{j=1}^{i} \frac{p_{(j)}}{s_j}$$

Using this equation we rewrite the objective function:

$$\begin{aligned}\sum_{i=1}^{n} C_i &= \sum_{i=1}^{n} C_{(i)} \\ &= \sum_{i=1}^{n}\sum_{j=1}^{i} \frac{p_{(j)}}{s_j} \\ &= \sum_{i=1}^{n} \frac{(n+1-i)}{s_i} p_{(i)}\end{aligned}$$

We define $q_i = \frac{n+1-i}{s_i}$, then q_i is a constant and we rewrite the objective function as:

$$\sum_{i=1}^{n} C_i = \sum_{i=1}^{n} q_i p_{(i)}.$$

We develop an algorithm that assigns all jobs to a position, such that the resulting schedule is optimal.

Algorithm 4.1 (Greedy appointment)**.** Let I be an instance of JSMD($w_j = 1$) without preemption and suppose without loss of generality that $p_1 \leq \cdots \leq p_n$.

1. Initialize: let $q_i = \frac{n+1-i}{s_i}$.
2. Assign job j to the position with the j'th largest q_i value.

Now we have to prove that this algorithm results in an optimal solution.

Theorem 4.2. *Algorithm 4.1 results in an optimal schedule for JDMS(*$w_j = 1$*) without preemption.*

Proof. Let σ be an optimal schedule for an instance of JDMS($w_j = 1$) without preemption and let σ' be a schedule obtained by Algorithm 4.1. Suppose σ differs from σ' and let k be the position where σ and σ' differ with largest q_k. Suppose that in σ job a is assigned to position k and in σ' job b is assigned to position k. Then the positions of both job a and b in σ differ from their position in σ'. According to the algorithm job b should be assigned to the largest remaining q_j. Thus $p_b \leq p_a$, and $q_i \leq q_k$. We make σ^* from σ, by switching job a and b and leaving the other jobs on the same position. Then the following holds:

$$\begin{aligned}\sum_{i=1}^{n} C_i^{\sigma} - \sum_{i=1}^{n} C_i^{\sigma^*} &= \sum_{i=1}^{n} q_i p_{(i)}^{\sigma} - \sum_{i=1}^{n} q_i p_{(i)}^{\sigma^*} \\ &= q_k p_a + q_i p_b - q_k p_b - q_i p_a \\ &= (q_k - q_i)(p_a - p_b) \\ &\geq 0\end{aligned}$$

Thus:

$$\sum_{i=1}^{n} C_i^{\sigma} \geq \sum_{i=1}^{n} C_i^{\sigma^*}.$$

As σ was already optimal, σ^* is optimal too. Repeating this process at most k times will give us schedule σ'. Therefore σ' is optimal as well. Thus Algorithm 4.1 results in an optimal schedule. $\square$

Note that when the speed function is constant this problem equals $1||\sum C_j$ and Algorithm 4.1 equals the SPT-algorithm.

4.2 JDMS($p_j = 1$) without preemption

We now restrict to unit processing times. We write $w_{(j)}$ to describe the weight of the job that is processed on position j. Again we rewrite the objective function:

$$w_{(i)}C_{(i)} = w_{(i)} \sum_{j=1}^{i} \frac{1}{s_j}.$$

And thus:

$$\sum_{i=1}^{n} w_{(i)}C_{(i)} = \sum_{i=1}^{n} w_{(i)} \sum_{j=1}^{i} \frac{1}{s_j}$$

We define coefficients $q_i = \sum_{j=1}^{i} \frac{1}{s_j}$. Note that $q_1 < \cdots < q_n$. The objective function can be written as:

$$\sum_{i=1}^{n} w_i C_i = \sum_{i=1}^{n} w_{(i)}C_{(i)} = \sum_{i=1}^{n} w_{(i)} q_i$$

This sum is minimized when the smallest weight is assigned to the largest coefficient, etc. Thus when we schedule the jobs in order of non-increasing weights we obtain an optimal schedule. Note that this order coincides with the WSPT rule.

4.3 JDMS without preemption

Lastly we have a look at JDMS without preemption. Again we rewrite the sum of completion times:

$$w_{(i)}C_{(i)} = w_{(i)} \sum_{j=0}^{i} \frac{p_{(j)}}{s_j}.$$

And thus:

$$\begin{aligned}
\sum_{i=1}^{n} w_i C_i &= \sum_{i=1}^{n} w_{(i)} C_{(i)} \\
&= \sum_{i=1}^{n} w_{(i)} \sum_{j=0}^{i} \frac{p_{(j)}}{s_j} \\
&= \sum_{1 \le j \le i \le n} \frac{w_{(i)} p_{(j)}}{s_j} \\
&= \sum_{1=j}^{n} \frac{p_{(j)}}{s_j} \left(\sum_{i=j}^{n} w_{(i)} \right)
\end{aligned}$$

Smith proved that WSPT order is optimal when the machine speed does not change [22], but unfortunately it can be arbitrarily bad compared to the optimum when the machine speed depends on the number of active jobs. This can be shown by only using two jobs:

Example 4.3 (WSPT may be arbitrarily bad)**.** *Suppose we have* 2 *jobs with* $p_1 = 2, p_2 = A, w_1 = 1, w_2 = A, s_1 = 1$ *and* $s_2 = A$*. If we use WSPT order job 2 proceeds job 1, as* $w_1/p_1 = \frac{1}{2} < 1 = w_2/p_2$*. The total weighted completion time is then:*

$$w_1 C_1 + w_2 C_2 = A \cdot \frac{A}{1} + 1 \cdot \left(\frac{A}{1} + \frac{2}{A} \right) = A^2 + A + \frac{2}{A}.$$

When we first complete job 1 and then complete job 2 *we get total weighted completion time:*

$$w_1 C_1 + w_2 C_2 = 1 \cdot \frac{2}{1} + A \cdot \left(\frac{2}{1} + \frac{A}{A} \right) = 3A + 2.$$

Thus the optimal solution has value $3A + 2$*. We compare the value of WSPT to the optimal value:*

$$\lim_{A \to \infty} \left(\frac{WSPT}{OPT} \right) = \lim_{A \to \infty} \left(\frac{A^2 + A + \frac{2}{A}}{3A + 2} \right) = \infty.$$

Thus WSPT order can be arbitrarily bad compared to the optimum.

Chapter 5

Unit weight JDMS

In this chapter we discuss the problem JDMS($w_j = 1$). Note that $1|prmpt|\sum C_j$ is a special case of JDMS($w_j = 1$). As we can preempt an infinite number of times, this is equivalent with dividing the machine capacity amongst jobs. In this chapter we show how to construct an optimal solution for this problem.

Before we give such a construction, we will prove several properties of an optimal solution. In this chapter we always assume that we work with n jobs, such that $p_1 \leq \cdots \leq p_n$, unless stated otherwise.

5.1 Properties of JDMS($w_j = 1$)

Lemma 5.1. *There is an optimal schedule in which $C_1 \leq \cdots \leq C_n$.*

Proof. Suppose we have an optimal schedule σ and it does not hold that $C_1 \leq \cdots \leq C_n$. Then we look at the smallest i such that $C_{i+1} < C_i$. We change σ to σ^* by only changing σ at job i and $i+1$, by processing job i such that it is finished at time C_{i+1}, so we have $C_i^{\sigma^*} = C_{i+1}$. This is possible as $p_i \leq p_{i+1}$. We show we can let job $i+1$ finish at time C_i without changing something about the other jobs.

As the time points where jobs finish stay the same, it holds that the speed of the machine is equal in σ and σ^* at every point in time. Therefore per time unit the same amount of work can be done in σ and σ^*. So there is exactly enough space for job $i+1$ to be finished at time C_i. Thus $\sum_{i=1}^n C_i = \sum_{i=1}^n C_i^{\sigma^*}$ and σ^* will remain optimal.

Iterating this process implies that there is an optimal schedule σ' such that $C_1 \leq \cdots \leq C_n$. $\square$

Using the structure of Lemma 5.1, we formulate JDMS($w_j = 1$) as a linear program. Therefore we introduce variables $\Delta_1, \cdots, \Delta_n$, where:

$$\Delta_i = \begin{cases} C_1 & \text{if } i = 1 \\ C_i - C_{i-1} & \text{if } 1 < i \leq n \end{cases}$$

At the interval $[0, C_1]$, the machine is operating at speed s_1 and during the intervals $[C_{i-1}, C_i]$, the machine is operating at speed s_i.

We rewrite the constraints for a feasible solution in linear equations with only variables $\Delta_1, \ldots, \Delta_n$. The sum of completion times can be rewritten as:

$$\sum_{i=1}^{n} C_i = \sum_{i=1}^{n} \sum_{j=1}^{i} \Delta_j = \sum_{i=1}^{n} (n - i + 1)\Delta_i.$$

To make sure the requested order on the completion times is enforced, we want $\Delta_i \geq 0$ for all i. Lastly we want to make sure that before time C_i, at least jobs $1, \ldots, i$ have been fully processed already. Thus:

$$\sum_{k=1}^{i} \Delta_k \cdot s_k \geq \sum_{k=1}^{i} p_k.$$

Combining these equations we obtain the following LP:

$$\begin{array}{rll} \text{minimize} & \displaystyle\sum_{i=1}^{n} (n - i + 1) \cdot \Delta_i & \\ \text{subject to} & \displaystyle\sum_{k=1}^{i} \Delta_k \cdot s_k \geq \sum_{k=1}^{i} p_k & , 1 \leq i \leq n \\ & \Delta_i \geq 0 & , 1 \leq i \leq n \end{array}$$

Note that a feasible solution for this LP does not correspond with an unique schedule, but with a non-empty set of schedules for which the completion times are set.

As we can formulate JDMS($w_j = 1$) as an LP, we can find an optimal solution in polynomial time by solving the LP. Though there is a another, faster, way to construct the optimal solution, which we will explain below.

We first prove that there exists an optimal solution, such that at all completion times, each job that has started is finished. To prove this property we introduce some additional notation. We define $y_i^\sigma(t)$ as the amount of processing that job i attained up to time t in schedule σ. We drop the σ from this notation if it is clear from the context.

Lemma 5.2. *We can find an optimal schedule such that for all i, j, where $1 \leq i, j \leq n$, it holds that $y_i(C_j) \in \{0, p_i\}$.*

Proof. We prove this lemma by using the LP formulation for JDMS($w_j = 1$). We reformulate the lemma in terms of this linear program. For each job one of the following has to hold:

- $\Delta_{k+1} = 0$, and thus $C_k = C_{k+1}$.
- If $C_{k+1} > C_k$, then it has to hold that $\sum_{j=1}^{k} s_j \Delta_j = \sum_{j=1}^{k} p_j$. If that is not the case, then, when we use Lemma 5.1 in combination with the assumption that a machine is always working at full speed, there exists a job $l \geq k+1$ such that $0 \leq y_l(C_k) \leq p_l$.

Thus we want to prove that for $k = 1, \dots, n-1$ either:

$$\Delta_{k+1} = 0 \quad (1) \qquad \text{or} \qquad \sum_{j=1}^{k} s_j \Delta_j = \sum_{j=1}^{k} p_j. \quad (2)$$

Suppose the contrary and we have an optimal solution such that there exists a k with:

$$\Delta_{k+1} > 0 \qquad \text{and} \qquad \sum_{j=1}^{k} s_j \Delta_j > \sum_{j=1}^{k} p_j.$$

We define ℓ as:

$$\ell = \max\{j \leq k | \Delta_j > 0\}.$$

We define two new feasible solutions, for some $\epsilon > 0$:

1. We define σ' as the solution where $\Delta_\ell = \Delta_\ell - \frac{\epsilon}{s_\ell}$ and $\Delta_{k+1} = \Delta_{k+1} + \frac{\epsilon}{s_{k+1}}$. We can lower Δ_ℓ because $\sum_{j=1}^{l} s_j \Delta_j > \sum_{j=1}^{k} p_j$. We see ϵ as the amount of processing time of jobs p_j with $j \geq k+1$ that in σ is processed before C_ℓ and in σ' is processed after C_k. The change in the objection value is:

$$\begin{aligned} \sum_{i=1}^{n} C_i^{\sigma'} - \sum_{i=1}^{n} C_i &= \sum_{i=1}^{n} (n-i+1)\Delta_i - \sum_{i=1}^{n} (n-i+1)\Delta_i \\ &= (n-k)\frac{\epsilon}{s_{k+1}} - (n-\ell+1)\frac{\epsilon}{s_\ell} \end{aligned}$$

2. We define σ'' as the solution where $\Delta_\ell^{\sigma''} = \Delta_\ell + \frac{\epsilon}{s_\ell}$ and $\Delta_{k+1}^{\sigma''} = \Delta_{k+1} - \frac{\epsilon}{s_{k+1}}$. We can lower Δ_{k+1} because $\Delta_{k+1} > 0$. We see ϵ as the amount of processing time of

jobs p_j with $j \geq k+1$ that in σ is processed after C_ℓ and in σ'' is processed before C_k. The change in the objection value is:

$$\begin{aligned}\sum_{i=1}^{n} C_i^{\sigma''} - \sum_{i=1}^{n} C_i &= \sum_{i=1}^{n}(n-i+1)\Delta_i^{\sigma''} - \sum_{i=1}^{n}(n-i+1)\Delta_i \\ &= (n-\ell+1)\frac{\epsilon}{s_\ell} - (n-k)\frac{\epsilon}{s_{k+1}}\end{aligned}$$

The change in the objective value is opposite for both new solutions. As σ was optimal, the change in the objective value has to be 0, and thus σ' and σ'' are both optimal as well.

Suppose k is the smallest value such that neither (1) or (2) holds. If we define:

$$\epsilon = \min\{\Delta_{k+1}, \sum_{j=1}^{k} s_j\Delta_j - \sum_{j=1}^{k} p_j\}.$$

Then either σ' or σ'' will give a feasible, optimal solution where either (1) or (2) holds for job k. We repeat this procedure until this property holds for all $k \in \{0\dots n\}$.

Thus if $C_1 \leq \cdots \leq C_n$, we can find an optimal schedule such that for all i, j, where $1 \leq i, j \leq n$, it holds that $y_i(C_j) \in \{0, p_i\}$. $\square$

Lemma 5.2 implies that we can divide the jobs into groups of consecutive jobs, such that all jobs in a group will start and end at the same time. We use G_i to denote the i'th group of jobs and we denote an optimal solution as $[G_1, \dots, G_k]$ (Figure 5.1).

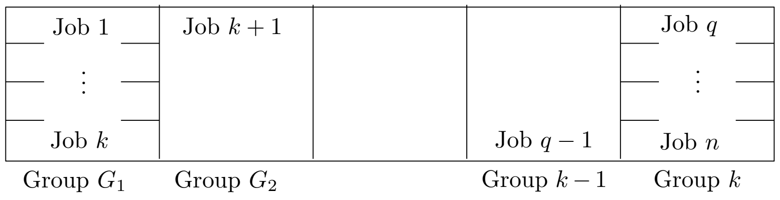

Figure 5.1: Schedule in which the jobs are divided into k groups.

5.2 Optimal algorithms

We give two algorithms that find an optimal schedule for this problem. Suppose we have n jobs with processing times $p_1 \leq \cdots \leq p_n$. Then, according to Lemma 5.1, we know there exists a optimal schedule such that $C_1 \leq \cdots \leq C_n$. According to Lemma 5.2 there should be an optimal schedule in which for all i, j, where $1 \leq i, j \leq n$, it holds that

$y_i(C_j) \in \{0, p_i\}$. So the only thing we need to know is how to partition the jobs into groups of consecutive jobs. We give two algorithms that will result in an optimal schedule.

5.2.1 Shortest path on a directed acyclic graph

Here we describe an algorithm that uses a shortest path algorithm in directed, acyclic graphs.

Algorithm 5.3 (Construction using graphs)**.** We construct a directed graph, $G = (V, A)$, where $V = \{1, \dots n+1\}$ is the set of vertices and A the set of edges. In V the jobs are represented by $\{1, \dots n\}$, and the last, $n+1$'st vertex represents the fact that all jobs have been scheduled. The set A of edges consists of all edges (i, j) with $i < j$, where an edge (i, j) corresponds to the group of jobs $i, \dots, j-1$. Processing all jobs in a group $i, \dots, j-1$, takes $\Delta_i = \left(\sum_{k=i}^{j-1} p_k\right)/s_i$ time. The length of edge (i, j) is set to $(n-i+1)\Delta_i$. Now every path from 1 to $n+1$ divides the jobs into groups, and the length of this path will correspond to the sum of completion times.

A shortest path algorithm for a directed acyclic graph takes $O(n^2)$ time, where n is the number of jobs.

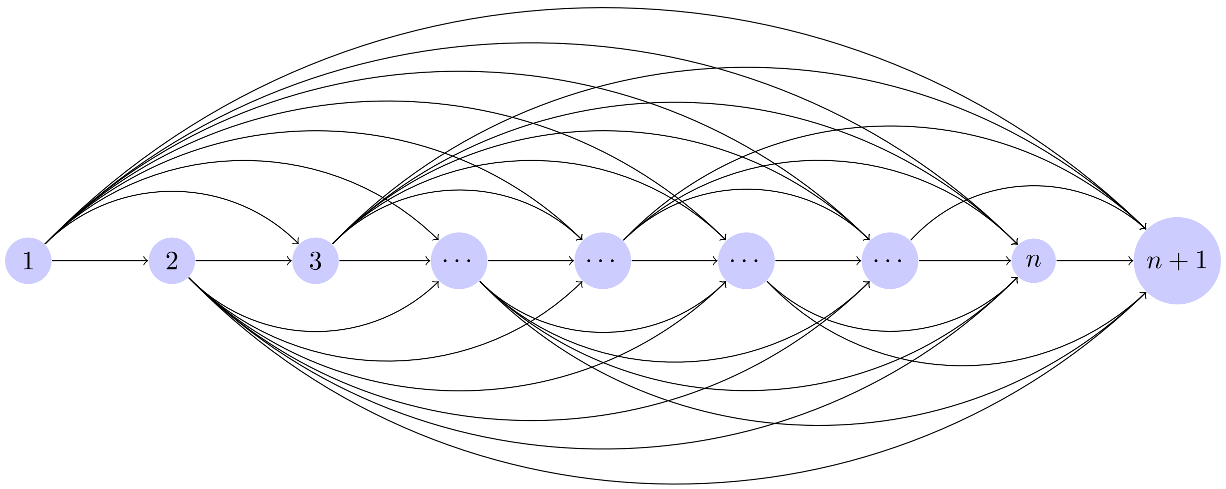

Figure 5.2: Visualized graph.

Lemma 5.4. *The length of a $1-(n+1)$ path in the graph $G=(V,A)$ as described in Algorithm 5.3 equals the sum of completion times of the schedule that the path represents.*

Proof. Suppose our path consists of the edges $(a_1,a_2),\dots,(a_{m-1},a_m)$. If $i \notin \{a_1,\dots a_m\}$, then $\Delta_i = 0$, as all jobs in the group are processed at the same time.

$$\begin{aligned}\sum_{j=1}^{m-1} w_{(a_j,a_{j+1})} &= \sum_{j=1}^{m}(n-a_j+1)\Delta_{a_j} \\ &= \sum_{i=1}^{n}(n-i+1)\Delta_i \\ &= \sum_{i=1}^{n} C_j\end{aligned}$$

Thus the length of a path in graph G, made according to Algorithm 5.3, equals the sum of completion times of the schedule that the path represents. □

Theorem 5.5. *Algorithm 5.3 results in an optimal schedule for JDMS($w_j = 1$)*

Proof. Clearly every path represents a way the groups could have been split up. According to Lemma 5.4, the length of the path equals the sum of completion times of the schedule that the path represents. Finding the shortest path will therefore give us a partition of the jobs that is a feasible solution to our problem, with minimal sum of completion times.

Every path corresponds to a schedule and every schedule satisfying Lemma 5.2 corresponds to a path. The length of the path is equal to the sum of completion times. Thus the shortest path corresponds to an optimal solution. □

5.2.2 Greedy algorithm

In this subsection we develop a greedy algorithm that finds an optimal schedule. We first give a more intuitive explanation and after that a formal definition of the algorithm in pseudo code.

Our greedy algorithm determines a feasible sequence of groups. This sequence indicates which groups of jobs should be processed in which order. Our algorithm first orders the jobs such that $p_1 \le p_2 \le \dots \le p_n$, which takes $O(n \log n)$ time. It then determines for each job i, whether it is better to schedule this job in the previous group or to start a new group for job i. When inequality (5.1) holds, job i is scheduled to be processed in the previous group and otherwise a new group will be started. The pseudo code of this greedy algorithm can be found in Algorithm 5.6.

$$\frac{|G_k| + (n+1-i)}{n+1-i} \le \frac{s_{G_k}}{s_i}. \tag{5.1}$$

Here we use s_{G_k} to refer to the speed at which G_k is processed.

Algorithm 5.6 (Greedy algorithm)**.** Suppose we want to schedule n jobs with processing times $p_1 \leq \cdots \leq p_n$. We schedule the jobs one by one, according to the following algorithm:

```
Initialization: $G_1 = \{1\}, k = 1, E = 1, s = s_1$

For $i = 2$ until $i = n$ do
{   If $(E + n + 1 - i)s_i \leq (n + 1 - i)s$
    Then
        $E \rightarrow E + 1$,
        $G_k \rightarrow G_k \cup \{i\}$
    Else
        $E \rightarrow 1$,
        $s \rightarrow s_i$,
        $G_{k+1} = \{i\}$
        $k \rightarrow k + 1$,
Return $[G_1, \ldots, G_k]$
```

This pseudo code takes $O(n)$ time.

Theorem 5.7. *Algorithm 5.6 finds an optimal partition of the jobs into groups.*

Proof. Suppose σ is an optimal schedule for an instance of this problem with n jobs, where $p_1 \leq \cdots \leq p_n$, such that $C_1 \leq \cdots \leq C_n$ (Lemma 5.1). Let σ^* be the solution according to Algorithm 5.6. We will show that the total completion time of σ^* is not more than that of σ, and thus σ^* is also an optimal schedule. Let job i be the first job in σ that is not scheduled according to Algorithm 5.6. Then there are two possible situations:

1. In σ, job i is scheduled in a new group G_k, whereas in σ^* it is still processed in group G_{k-1}.
2. Job i is scheduled in the previous group G_k, while it should be scheduled in a new group.

Suppose we are in the first situation: in σ, job i is scheduled in a new group G_k, whereas in σ^* it is still processed in group G_{k-1}. Then we change σ to σ' by merging G_k and G_{k-1}. Then in σ', jobs $1, \ldots, i$ are scheduled the same as in σ^*.

We look at the total value that groups G_k, G_{k-1} and $G_k \cup\ G_{k-1}$ contribute to the objective value. These are the only jobs that are interesting to look at, as the value that the other jobs contribute to the objective value in σ and σ' does not change.

Given some schedule τ, let $c_\tau(S) = \sum_{j\in S}(n+1-j)\Delta_j(\tau)$ be the function that computes the value that group S in τ contributes to the objective value. Then:

$$c_\sigma(G_k) + c_\sigma(G_{k-1}) = (|G_{k-1}| + |G_k| + r)\frac{\sum_{i\in G_{k-1}} p_i}{s_{G_{k-1}}} + (|G_k) + r|\frac{\sum_{i\in G_k} p_i}{s_{G_k}}. \tag{5.2}$$

$$c_{\sigma'}(G_k \cup\ G_{k-1}) = (|G_{k-1}| + |G_k| + r)\frac{\sum_{i\in G_k\cup\ G_{k-1}} p_i}{s_{G_{k-1}}}. \tag{5.3}$$

Let r denote the number of jobs scheduled after group G_k, which is equal in both σ and σ'.

According to Algorithm 5.6 it holds that:

$$\frac{|G_{k-1}| + (n+1-i)}{n+1-i} \leq \frac{s_{G_{k-1}}}{s_{G_k}}. \tag{5.4}$$

Combining the fact that $|G_k| + r = n+1-i$ and (5.4) we know that:

$$\frac{|G_{k-1}| + |G_k| + r}{|G_k| + r} \leq \frac{s_{G_{k-1}}}{s_{G_k}}. \tag{5.5}$$

Rewrite (5.5):

$$\frac{(|G_{k-1}| + |G_k| + r)}{s_{G_{k-1}}} \leq \frac{(|G_k| + r)}{s_{G_k}}. \tag{5.6}$$

Multiplying both sides with $\sum_{i\in G_k} p_i$:

$$(|G_{k-1}| + |G_k| + r)\frac{\sum_{i\in G_k} p_i}{s_{G_{k-1}}} \leq (|G_k| + r)\frac{\sum_{i\in G_k} p_i}{s_{G_k}}. \tag{5.7}$$

Adding $(|G_{k-1}| + |G_k| + r)\frac{\sum_{i\in G_{k-1}} p_i}{s_{G_{k-1}}}$ yields:

$$(|G_{k-1}|+|G_k|+r)\frac{\sum_{i\in G_k\cup\ G_{k-1}} p_i}{s_{G_{k-1}}} \leq (|G_{k-1}|+|G_k|+r)\frac{\sum_{i\in G_{k-1}} p_i}{s_{G_{k-1}}} + (|G_k|+r)\frac{\sum_{i\in G_k} p_i}{s_{G_k}}. \tag{5.8}$$

Combining (5.2), (5.3) and (5.8) yields:

$$c_{\sigma'}(G_k \cup\ G_{k-1}) \leq c_\sigma(G_k) + c_\sigma(G_{k-1}). \tag{5.9}$$

Thus the value of our changed schedule σ' is smaller or equal than the objective value of σ, thus σ' is optimal as well.

Now suppose we have the second situation: job i is scheduled in the previous group G_k, while it should be scheduled in a new group. Then we change σ to σ' by splitting group G_k in two groups at job i: $G_{k,1}$ and $G_{k,2}$. Job i is now the first job in group $G_{k,2}$. Then in σ', jobs $1, \ldots, i$ are scheduled the same as in σ^*.

We look at the total value that groups G_k, $G_{k,1}$ and $G_{k,2}$ contribute to the objective value. These are the only jobs that are interesting to look at, as the value that the other jobs contribute to the objective value in σ and σ' does not change.

$$c_\sigma(G_k) = (|G_k| + r)\frac{\sum_{i \in G_k} p_i}{s_{G_k}}. \tag{5.10}$$

$$c_{\sigma'}(G_{k,1}) + c_{\sigma'}(G_{k,2}) = (|G_{k,1}| + |G_{k,2}| + r)\frac{\sum_{i \in G_{k,1}} p_i}{s_{G_{k,1}}} + (|G_{k,2}| + r)\frac{\sum_{i \in G_{k,2}} p_i}{s_i}. \tag{5.11}$$

Let r be the number of jobs scheduled after G_k.

According to Algorithm 5.6 it holds that:

$$\frac{|G_{k,1}| + (n + 1 - i)}{n + 1 - i} > \frac{s_{G_{k,1}}}{s_i}. \tag{5.12}$$

Combining the fact that $|G_{k,2}| + r = n + 1 - i$ and (5.12) we know that:

$$\frac{|G_{k,1}| + |G_{k,2}| + r}{|G_{k,2}| + r} > \frac{s_{G_{k,1}}}{s_i}. \tag{5.13}$$

Rewrite (5.13):

$$\frac{(|G_{k,1}| + |G_{k,2}| + r)}{s_{G_{k,1}}} > \frac{(|G_{k,2}| + r)}{s_i}. \tag{5.14}$$

Multiplying both sides with $\sum_{i \in G_{k,2}} p_i$:

$$(|G_{k,1}| + |G_{k,2}| + r)\frac{\sum_{i \in G_{k,2}} p_i}{s_{G_{k,1}}} > (|G_{k,2}| + r)\frac{\sum_{i \in G_{k,2}} p_i}{s_i}. \tag{5.15}$$

Adding $(|G_{k,1}| + |G_{k,2}| + r)\frac{\sum_{i \in G_{k,1}} p_i}{s_{G_{k,1}}}$ yields:

$$(|G_{k,1}|+|G_{k,2}|+r)\frac{\sum_{i \in (G_{k,1} \cup\, G_{k,2})} p_i}{s_{G_{k,1}}} > (|G_{k,1}|+|G_{k,2}|+r)\frac{\sum_{i \in G_{k,1}} p_i}{s_{G_{k,1}}} + (|G_{k,2}|+r)\frac{\sum_{i \in G_{k,2}} p_i}{s_i}. \tag{5.16}$$

Rewrite (5.16):

$$(|G_k| + r)\frac{\sum_{i \in G_k} p_i}{s_{G_k}} > (|G_{k,1}| + |G_{k,2}| + r)\frac{\sum_{i \in G_{k,1}} p_i}{s_{G_{k,1}}} + (|G_{k,2}| + r)\frac{\sum_{i \in G_{k,2}} p_i}{s_i}. \tag{5.17}$$

Combining (5.10), (5.11) and (5.16) yields:

$$c(G_k) > c(G_{k,1}) + c(G_{k,2}). \tag{5.18}$$

Thus the value of our changed schedule σ' is smaller than the objective value of σ, contradicting the fact that σ is optimal. Therefore this situation cannot happen.

Every time we change an optimal schedule according to one of the two options above, schedule σ' is also optimal. On top of that we know that the index of the first job for which σ' makes a different decision than σ^* has increased compared to σ. Therefore, by changing the schedule at most n times, we find an optimal schedule. This is the same schedule as σ^*. $\square$

Chapter 6

JDMS with general weights

In this chapter we work with general weights instead of unit weights. It appears that this problem is in some ways similar to JDMS($w_j = 1$), and with only slight changes, some of the properties we have proven still hold. The main difference is that we cannot find an order on the completion times that guarantees an optimal solution. But when an order on the job completions is enforced, Lemma 5.2 still holds for the weighted problem, which we show in Lemma 6.1. Using this lemma, we propose a greedy algorithm, similar to Algorithm 5.6, that finds an optimal partition of the jobs into groups given a predetermined order of job completions.

6.1 Properties of JDMS

In Chapter 5 we proved that there exists an optimal solution such that for all i, j, where $1 \leq i, j \leq n$, it holds that $y_i(C_j) \in \{0, p_i\}$. We want to prove that this still holds for JDMS(w_j). We first prove that, whenever we enforce an order on the completion times, there exists a solution that satisfies this property. Note that as the optimal solution obeys a certain order of job completions, there exists an optimal solution such that for all i, j, where $1 \leq i, j \leq n$, it holds that $y_i(C_j) \in \{0, p_i\}$.

Suppose that the jobs are indexed such that the completion times satisfy $C_1 \leq \cdots \leq C_n$. We formulate JDMS as a linear program in the same way as we did in Chapter 5. Thus we again introduce variables $\Delta_1, \cdots, \Delta_n$, where:

$$\Delta_i = \begin{cases} C_1 & \text{if } i = 1 \\ C_i - C_{i-1} & \text{if } 1 < i \leq n \end{cases}$$

The LP is as follows:

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{n} \left(\sum_{j=i}^{n} w_j \cdot \Delta_i \right) \\
\text{subject to} \quad & \sum_{k=1}^{i} \Delta_k \cdot s_k \geq \sum_{j=1}^{i} p_j, \quad 1 \leq i \leq n \\
& \Delta_i \geq 0, \qquad\qquad\qquad 1 \leq i \leq n
\end{aligned}$$

Using this LP we prove the next lemma:

Lemma 6.1. *We can find an optimal schedule such that for all i, j, where $1 \leq i, j \leq n$, it holds that $y_i(C_j) \in \{0, p_i\}$.*

Proof. We prove this lemma using the same arguments as we used in Lemma 5.2. Given an order of completion times, we make a corresponding LP and reformulate this lemma in terms of this linear program. For each job $k = 1, \dots, n-1$ one of the following has to hold:

$$\Delta_{k+1} = 0 \quad (1) \qquad \text{or} \qquad \sum_{j=1}^{k} s_j \Delta_j = \sum_{j=1}^{k} p_j. \quad (2)$$

Suppose we have an optimal solution such that there exists a k with:

$$\Delta_{k+1} > 0 \qquad \text{and} \qquad \sum_{j=1}^{k} s_j \Delta_j > \sum_{j=1}^{k} p_j.$$

We define ℓ as:

$$\ell = \max\{j \leq k | \Delta_j > 0\}.$$

We define two new feasible solutions in the same way as we did in the proof of Lemma 5.2, for some $\epsilon > 0$:

1. We define σ' as the solution where $\Delta_\ell^{\sigma'} = \Delta_\ell^{\sigma} - \frac{\epsilon}{s_\ell}$ and $\Delta_{k+1}^{\sigma'} = \Delta_{k+1}^{\sigma} + \frac{\epsilon}{s_{k+1}}$. The change in the objection value is:

$$\begin{aligned}
\sum_{i=1}^{n} w_i C_i^{\sigma'} - \sum_{i=1}^{n} w_i C_i^{\sigma} &= \sum_{1 \leq i \leq j \leq n} w_j \Delta_i^{\sigma'} - \sum_{1 \leq i \leq j \leq n} w_j \Delta_i^{\sigma} \\
&= \left(\sum_{i=k+1}^{n} w_i \right) \frac{\epsilon}{s_{k+1}} - \left(\sum_{i=\ell}^{n} w_i \right) \frac{\epsilon}{s_\ell}.
\end{aligned}$$

2. We define σ'' as the solution where $\Delta_\ell^{\sigma''} = \Delta_\ell + \frac{\epsilon}{s_\ell}$ and $\Delta_{k+1}^{\sigma''} = \Delta_{k+1} - \frac{\epsilon}{s_{k+1}}$. The change in the objection value is:

$$\begin{aligned}\sum_{i=1}^{n} w_i C_i^{\sigma''} - \sum_{i=1}^{n} w_i C_i^{\sigma} &= \sum_{1\le i\le j\le n} w_j \Delta_i^{\sigma''} - \sum_{1\le i\le j\le n} w_j \Delta_i^{\sigma} \\ &= \left(\sum_{i=\ell}^{n} w_i\right)\frac{\epsilon}{s_\ell} - \left(\sum_{i=k+1}^{n} w_i\right)\frac{\epsilon}{s_{k+1}}.\end{aligned}$$

As:

$$\sum_{i=1}^{n} w_i C_i(\sigma') - \sum_{i=1}^{n} w_i C_i(\sigma) = -\left(\sum_{i=1}^{n} w_i C_i(\sigma'') - \sum_{i=1}^{n} w_i C_i(\sigma)\right),$$

at least one of the two solutions is better than or equal to σ. As σ is optimal, we actually know that both new solutions are also optimal.

Suppose k is the smallest value such that neither (1) or (2) holds. Let:

$$\epsilon = \min\{\Delta_{k+1}, \sum_{j=1}^{k} s_j \Delta_j - \sum_{j=1}^{k} p_j\},$$

then either σ' or σ'' will give a optimal solution where either (1) or (2) holds for job k. We repeat this procedure until this property holds for all $k \in \{0 \dots n\}$.

When the completion times satisfy $C_1 \le \cdots \le C_n$, we can find an optimal schedule such that for all i, j, where $1 \le i, j \le n$ it holds that $y_i(C_j) \in \{0, p_i\}$. As this holds for any order, this will also hold for the order of some optimal solution. Therefore there exists an optimal solution such that or all i, j, where $1 \le i, j \le n$, it holds that $y_i(C_j) \in \{0, p_i\}$. □

6.2 Greedy algorithm

In this section, we design a greedy algorithm that, given an order on the completion times, finds an optimal schedule for the weighted version of JDMS. According to Lemma 6.1 there should be an optimal schedule in which for all i, j, where $1 \le i, j \le n$ it holds that $y_i(C_j) \in \{0, p_i\}$. So, similar to unweighted JDMS, the only thing we need to know is in which groups we have to split up the jobs. We give a greedy algorithm that results in an optimal schedule for the requested order. We first give a more intuitive explanation and formal definition of the algorithm in pseudo code after.

Algorithm 6.2 returns a feasible sequence of groups. This sequence indicates which groups of jobs should be processed in what order. Our algorithm first orders the jobs such that $w_1 \ge w_2 \ge \cdots \ge w_n$, which takes $O(n \log n)$ time. It then determines for each job i, whether it is better to schedule this job in the previous group or to start a new group for job i. When inequality (6.1) holds, then job i is scheduled to be processed in the previous group. Otherwise a new group will be started. The pseudo code of this greedy algorithm can be found in Algorithm 6.2.

$$\frac{\sum_{j\in G_k} w_j + \sum_{j=i}^n w_j}{\sum_{j=i}^n w_j} \leq \frac{s_{G_k}}{s_i}. \tag{6.1}$$

Here we use s_{G_k} to refer to the speed at which G_k is processed.

Note that when $w_j = 1$ for all j, we have a special case of JDMS. In that case inequality (6.1) equals inequality (5.1).

Algorithm 6.2 (Greedy algorithm)**.** Suppose we want to schedule n jobs with processing times $p_1 \leq \cdots \leq p_n$. We schedule the jobs one by one, according to the following algorithm:

```
Initialization: $G_1 = \{1\}, k = 1, E = w_1, s = s_1$
For $i = 2$ until $i = n$ do
  If $(E + \sum_{j=i}^n w_j)s_i \leq (\sum_{j=i}^n w_j)s$
    Then
      $E \to E + w_i$,
      $G_k \to G_k \cup \{i\}$
    Else
      $E \to w_i$,
      $s \to s_i$,
      $G_{k+1} \to \{i\}$
      $k \to k + 1$,
Return $[G_1, \dots, G_k]$
```

Theorem 6.3. *When an order on the completion times is enforced Algorithm 6.2 results in an optimal schedule for JDMS.*

Proof. Suppose σ is an optimal schedule for an instance of JDMS with n jobs satisfying $C_1 \leq \cdots \leq C_n$. Let σ^* be the solution according to the algorithm. Then we want to show that a schedule according to the algorithm will give a solution with an equal objective value. Let job i be the first job in σ that is not scheduled according to the algorithm. Then there are two possible situations:

1. In σ, job i is scheduled in a new group G_k, whereas in σ^* it is still processed in group G_{k-1}.
2. Job i is scheduled in the previous group G_k, while it should be scheduled in a new group.

Suppose we are in the first situation: in σ, job i is scheduled in a new group G_k, whereas in σ^* it is still processed in group G_{k-1}. Then we change σ to σ' by merging G_k and G_{k-1}. Then in σ' , jobs $1, \ldots, i$ are scheduled the same as in σ^*.

We look at the total value that groups G_k, G_{k-1} and $G_k \cup\ G_{k-1}$ contribute to the objective value. The jobs in these groups are the only jobs that are interesting to look at, as the value that the other jobs contribute to the objective value in σ and σ' does not change. Let c be the function that computes the value that a group contributes to the objective value. Then:

$$c_\sigma(G_k)+c_\sigma(G_{k-1}) = \left(\sum_{j\in G_{k-1}} w_j + \sum_{j\in G_k} w_j + r\right) \frac{\sum_{j\in G_{k-1}} p_j}{s_{G_{k-1}}} + \left(\sum_{j\in G_k} w_j + r\right) \frac{\sum_{j\in G_k} p_j}{s_{G_k}}, \tag{6.2}$$

$$c_{\sigma'}(G_k \cup\ G_{k-1}) = \left(\sum_{j\in G_{k-1}} w_j + \sum_{j\in G_k} w_j + r\right) \frac{\sum_{j\in G_k \cup\ G_{k-1}} p_j}{s_{G_{k-1}}}. \tag{6.3}$$

Let r be the number of jobs scheduled after G_k.

According to Algorithm 6.2 it should hold that:

$$\frac{\sum_{j\in G_{k-1}} w_j + \sum_{j=i}^{n} w_j}{\sum_{j=i}^{n} w_j} \leq \frac{s_{G_k}}{s_i}. \tag{6.4}$$

Combining the fact that $\sum_{j\in G_k} w_j + r = \sum_{j=i}^{n} w_j$ and (6.4) we know that:

$$\frac{\sum_{j\in G_{k-1}} w_j + \sum_{j\in G_k} w_j + r}{\sum_{j\in G_k} w_j + r} \leq \frac{s_{G_k}}{s_i}. \tag{6.5}$$

We rewrite (6.5):

$$\frac{\sum_{j\in G_{k-1}} w_j + \sum_{j\in G_k} w_j + r}{s_{G_{k-1}}} \leq \frac{\sum_{j\in G_k} w_j + r}{s_{G_k}}. \tag{6.6}$$

Multiplying both sides with $\sum_{j\in G_k} p_j$:

$$\left(\sum_{j\in G_{k-1}} w_j + \sum_{j\in G_k} w_j + r\right) \frac{\sum_{j\in G_k} p_j}{s_{G_{k-1}}} \leq \left(\sum_{j\in G_k} w_j + r\right) \frac{\sum_{j\in G_k} p_j}{s_{G_k}}. \tag{6.7}$$

Adding $\left(\sum_{j\in G_{k-1}} w_j + \sum_{j\in G_k} w_j + r\right) \frac{\sum_{j\in G_{k-1}} p_j}{s_{G_{k-1}}}$ yields:

$$\left(\sum_{j\in G_{k-1}} w_j + \sum_{j\in G_k} w_j + r\right) \frac{\sum_{j\in G_k\cup\, G_{k-1}} p_j}{s_{G_{k-1}}} \leq \left(\sum_{j\in G_k} w_j + r\right) \frac{\sum_{j\in G_k} p_j}{s_{G_k}} + \left(\sum_{j\in G_{k-1}} w_j + \sum_{j\in G_k} w_j + r\right) \frac{\sum_{j\in G_{k-1}} p_j}{s_{G_{k-1}}}. \tag{6.8}$$

Combining (6.2), (6.3) and (6.8) yields:

$$c_{\sigma'}(G_k \cup\, G_{k-1}) \leq c_\sigma(G_k) + c_\sigma(G_{k-1}). \tag{6.9}$$

Thus the value of our changed schedule σ' is smaller or equal than the objective value of σ, thus σ' is optimal as well.

Now suppose we have the second situation: job i is scheduled in the previous group G_k, while it should be scheduled in a new group. Then we change σ to σ' by splitting group G_k in two groups at job i: $G_{k,1}$ and $G_{k,2}$. Job i is now the first job in group $G_{k,2}$. Then in σ', jobs $1,\ldots,i$ are scheduled the same as in σ^*.

We look at the total value that groups G_k, $G_{k,1}$ and $G_{k,2}$ contribute to the objective value. The jobs in these groups are the only jobs that are interesting to look at, as the value that the other jobs contribute to the objective value in σ and σ' does not change.

$$c_\sigma(G_k) = \left(\sum_{j\in G_k} w_j + r\right) \frac{\sum_{j\in G_k} p_j}{s_{G_k}}, \tag{6.10}$$

$$c_{\sigma'}(G_{k,1}) + c_\sigma(G_{k,2}) = \left(\sum_{j\in G_{k,1}} w_j + \sum_{j\in G_{k,2}} w_j + r\right) \frac{\sum_{j\in G_{k,1}} p_j}{s_{G_{k,1}}} + \left(\sum_{j\in G_{k,2}} w_j + r\right) \frac{\sum_{i\in G_{k,2}} p_i}{s_i}. \tag{6.11}$$

Let r be the number of jobs scheduled after G_k.

According to Algorithm 6.2 it should hold that:

$$\frac{\sum_{j\in G_{k,1}} w_j + \sum_{j=i}^n w_j}{\sum_{j=i}^n w_j} > \frac{s_{G_{k,1}}}{s_i}. \tag{6.12}$$

Combining the fact that $\sum_{j\in G_{k,1}} w_j + r = \sum_{j=i}^n w_j$ and (6.12) we know that:

$$\frac{\sum_{j\in G_{k,1}} w_j + \sum_{j\in G_{k,2}} w_j + r}{\sum_{j\in G_{k,2}} w_j + r} > \frac{s_{G_{k,1}}}{s_i}. \tag{6.13}$$

We rewrite (6.13):

$$\frac{\sum_{j\in G_{k,1}} w_j + \sum_{j\in G_{k,2}} w_j + r}{s_{G_{k,1}}} > \frac{\sum_{j\in G_{k,2}} w_j + r}{s_i}. \tag{6.14}$$

Multiplying both sides with $\sum_{j\in G_{k,2}} p_j$:

$$\left(\sum_{j\in G_{k,1}} w_j + \sum_{j\in G_{k,2}} w_j + r\right)\frac{\sum_{j\in G_{k,2}} p_j}{s_{G_{k,1}}} > \left(\sum_{j\in G_{k,2}} w_j + r\right)\frac{\sum_{j\in G_{k,2}} p_j}{s_i}. \tag{6.15}$$

Adding $(\sum_{j\in G_{k,1}} w_j + \sum_{j\in G_{k,2}} w_j + r)\frac{\sum_{j\in G_{k,1}} p_j}{s_{G_{k,1}}}$ yields:

$$\begin{aligned}\left(\sum_{j\in G_{k,1}} w_j + \sum_{j\in G_{k,2}} w_j + r\right)\frac{\sum_{j\in G_{k,1}\cup\, G_{k,2}} p_j}{s_{G_{k,1}}} &> \left(\sum_{j\in G_{k,1}} w_j + \sum_{j\in G_{k,2}} w_j + r\right)\frac{\sum_{j\in G_{k,1}} p_j}{s_{G_{k,1}}} \\ &\quad + \left(\sum_{j\in G_{k,2}} w_j + r\right)\frac{\sum_{j\in G_{k,2}} p_j}{s_i}.\end{aligned} \tag{6.16}$$

We rewrite (6.17):

$$\begin{aligned}\left(\sum_{j\in G_k} w_j + r\right)\frac{\sum_{j\in G_{k,1}\cup\, G_{k,2}} p_j}{s_{G_{k,1}}} &> \left(\sum_{j\in G_{k,1}} w_j + \sum_{j\in G_{k,2}} w_j + r\right)\frac{\sum_{j\in G_{k,1}} p_j}{s_{G_{k,1}}} \\ &\quad + \left(\sum_{j\in G_{k,2}} w_j + r\right)\frac{\sum_{j\in G_{k,2}} p_j}{s_i}.\end{aligned} \tag{6.17}$$

Combining (6.10), (6.11) and (6.17) yields:

$$c_\sigma(G_k) > c_{\sigma'}(G_{k,1}) + c_{\sigma'}(G_{k,2}). \tag{6.18}$$

Thus the value of our changed schedule σ' is smaller than the objective value of σ, contradicting the fact that σ is optimal. Therefore this situation cannot happen.

Every time we change an optimal schedule according to one of the two options above, schedule σ' is also optimal. On top of that we know that the index of the first job for which σ' makes a different decision than σ^* has increased compared to σ. Therefore, by changing the schedule at most n times, we find an optimal schedule. This is the same schedule as σ^*.

□

Therefore, for any order, we can find an optimal schedule for that specific order. But which order will lead to an optimal solution?

6.3 Unit processing times: JDMS($p_j = 1$)

We first proof that when all jobs have unit processing times, thus $p_j = 1$ for all jobs j, we can find an order that leads to an optimal solution.

Lemma 6.4. *Suppose we have an instance of JDMS($p_j = 1$) such that $w_1 \geq \cdots \geq w_n$, then there is an optimal schedule in which $C_1 \leq \cdots \leq C_n$.*

Proof. Suppose we have an optimal schedule σ and it does not hold that $C_1^\sigma \leq \cdots \leq C_n^\sigma$. Then we look at the smallest i such that $C_{i+1}^\sigma < C_i^\sigma$ and $w_i > w_{i+1}$ (if $w_i = w_{i+1}$ then job i and $i+1$ are identical and therefore switching job i with job j will result in an optimal schedule as well). We change σ to σ^* by processing job $i+1$ whenever σ processes job i and job i whenever σ processes job $i+1$. All other jobs are processed as in σ. As $p_i = p_{i+1}$, we know that $C_{i+1}^{\sigma^*} = C_i^\sigma$ and $C_i^{\sigma^*} = C_{i+1}^\sigma$ and $C_j^{\sigma^*} = C_j^\sigma$ for all $j \neq i, i+1$.

Thus $w_j C_j^\sigma = w_j C_j^{\sigma^*}$ for all $j \neq i, i+1$. Furthermore, as $w_i > w_{i+1}$, $C_{i+1}^{\sigma^*} > C_i^{\sigma^*}$, $C_i^\sigma = C_{i+1}^{\sigma^*}$ and $C_{i+1}^\sigma = C_i^{\sigma^*}$, some simple rewriting learns us that:

$$\begin{aligned}
w_i C_i^\sigma + w_{i+1} C_{i+1}^\sigma &= w_{i+1} C_i^{\sigma^*} + w_i C_{i+1}^{\sigma^*} \\
&= w_{i+1} C_i^{\sigma^*} + w_i C_{i+1}^{\sigma^*} - (w_i C_i^{\sigma^*} + w_{i+1} C_{i+1}^{\sigma^*}) + (w_i C_i^{\sigma^*} + w_{i+1} C_{i+1}^{\sigma^*}) \\
&= (w_i - w_{i+1})(C_{i+1}^{\sigma^*} - C_i^{\sigma^*}) + (w_i C_i^{\sigma^*} + w_{i+1} C_{i+1}^{\sigma^*}) \\
&> w_i C_i^{\sigma^*} + w_{i+1} C_{i+1}^{\sigma^*}
\end{aligned}$$

Thus $\sum_{i=1}^n w_i C_i^\sigma > \sum_{i=1}^n w_i C_i^{\sigma^*}$

As σ is optimal it has to hold that $w_i = w_{i+1}$, which contradicts with the fact that $w_i > w_{i+1}$. Therefore σ could not be optimal. □

So when we number the jobs such that $w_1 \geq \cdots \geq w_n$, we know that there exists an optimal solution with $C_1 \leq \cdots \leq C_n$ and Algorithm 6.2 will find this solution.

6.4 Arbitrary processing times

Dropping the restriction of unit processing time may seem a small change, but unfortunately we cannot find an order that will lead to an optimal schedule. Smith [22] has proven that WSPT is optimal for the problem if the machine speed does not change.

Therefore, if there exists any way to number the jobs such that there always exist an optimal solution with $C_1 \leq \cdots \leq C_n$, it has to be according to the decreasing ratio w_j/p_j . Unfortunately, there does not always exist an optimal schedule for this order of jobs. We show this by giving a counterexample:

Example 6.5 (WSPT not optimal for decreasing speed)**.** *Suppose we have* 2 *jobs with* $p_1 = 1, p_2 = 10, w_1 = 1, w_2 = 9$*,* $s_1 = 5$ *and* $s_2 = 1$*, then it holds that* $w_1/p_1 > w_2/p_2$*. Therefore, according to WSPT, there should be an optimal solution with* $C_1 \geq C_2$*. According to Algorithm 6.2 it is optimal to schedule them at the same time.*

$$w_1C_1 + w_2C_2 = (9+1) \cdot \frac{10+1}{5} = 22$$

Suppose we process job 1 after job 2. Then the objective value is:

$$w_1C_1 + w_2C_2 = 1 \cdot 3 + 9 \cdot 2 = 21.$$

In this example we see that it can be strictly better to have $C_2 < C_1$ *and there does not always exists an optimal schedule that obeys the WSPT order.*

As WSPT-order doesn't lead to an optimal solution, we cannot guarantee that Algorithm 6.2 will find an optimal schedule. In fact, using WSPT-order can even lead to an arbitrary bad solution.

Example 6.6 (WSPT arbitrarily bad for increasing speeds)**.** *Suppose we have two jobs with* $p_1 = 2, p_2 = A, w_1 = 1, w_2 = A, s_1 = 1$ *and* $s_2 = A$*. If we use WSPT order, we want* $C_1 \leq C_2$*, as* $w_1/p_1 > w_2/p_2$*. According to algorithm 6.2 it is optimal to first complete job* 2 *and then complete job* 1*. Then we get objective value:*

$$w_1C_1 + w_2C_2 = A \cdot \frac{A}{1} + 1 \cdot \left(\frac{A}{1} + \frac{2}{A}\right) = A^2 + A + \frac{2}{A}$$

When we first complete job 1 and then complete job 2 *we get objective value:*

$$w_1C_1 + w_2C_2 = 1 \cdot \frac{2}{1} + A \cdot \left(\frac{2}{1} + \frac{A}{A}\right) = 3A + 2$$

Thus the optimal solution has at most value $3A + 2$*. By letting* A *go to infinity, the ratio also approaches infinity:*

$$\lim_{A \to \infty} \left(\frac{WSPT}{OPT}\right) \geq \lim_{A \to \infty} \left(\frac{A^2 + A + \frac{2}{A}}{3A + 2}\right) = \infty$$

Thus WSPT order can be arbitrarily bad compared to the optimum.

Chapter 7

JDMS with release dates

In this chapter we add release dates to JDMS. Note that when the machine speed is constant we have problem $1|r_j|\sum w_jC_j$, which is proven to be strongly NP-hard [14] (Section 3.1.4). Therefore JDMS(r_j) is strongly NP-hard as well.

When restricting the problem in other ways, interesting problems again occur. By restricting to unit weights for example: JDMS($r_j, w_j = 1$). The machine speed is $\overline{s}_i$, when there are i jobs available at that moment. We write $r_i(t)$ to denote the remaining processing time of job i at time t. We already know that SRPT is an optimal algorithm for $1|preempt, r_j|\sum C_j$, see Theorem 3.6. When the speed function $\overline{s}$ is non-decreasing, we can extend this result to JDMS($r_j, w_j = 1$).

Theorem 7.1. *If the speed function $\overline{s}$ is non-decreasing, then JDMS($r_j, w_j = 1$) can be solved in polynomial time by SRPT.*

Proof. Consider an optimal schedule σ in which available job i with the shortest remaining processing time is not being processed at time t, and instead available job j is being processed, so $r_i(t) < r_j(t)$. In total $r_i(t) + r_j(t)$ is spent on jobs i and j after time t.

Now we change σ to σ':

1. Take the first $r_i(t)$ units of time that were devoted to either of jobs i and j after time t, and use them instead to process job i to completion.
2. Take the remaining $r_j(t)$ units of time that were spent processing job i and j after time t and use them to schedule job j.

In Figure 7.1 this interchange argument is visualized. We obtain a better schedule σ', as $C_i^{\sigma'} < \min(C_i^{\sigma}, C_j^{\sigma})$ and $C_j^{\sigma'} \leq \max(C_i^{\sigma}, C_j^{\sigma})$. Furthermore, as $C_i^{\sigma'} < \min(C_i^{\sigma}, C_j^{\sigma})$, we can begin to process faster on a higher speed (as $\overline{s}$ is non-decreasing). Therefore $C_k^{\sigma'} \leq C_k^{\sigma}$

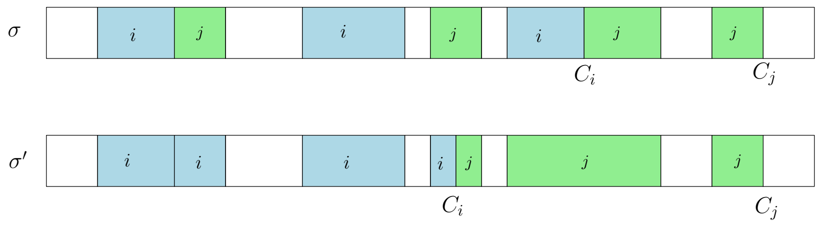

Figure 7.1: Visualized interchange argument

for all jobs k with $k \geq i$. As some jobs are completed earlier and no jobs are completed later, σ' is a better schedule than σ, which contradicts the optimality of σ. $\square$

For an arbitrary speed function $\overline{s}$ an optimal schedule may require unforced idleness. A non-delay schedule can be even arbitrary bad.

Theorem 7.2. *A non-delay schedule for JDMS($r_j, w_j = 1$) can be arbitrarily bad compared to the optimal solution.*

Proof. In a non-delay schedule we cannot leave a machine idle when there are jobs available for processing. We look at the following instance:

$p_1 = A$, $p_2 = A$,
$r_1 = 0$, $r_2 = 2$.
$\overline{s}_1 = 1$, $\overline{s}_2 = A$,

In Figure 7.2 are two possible schedules. The top one is the unique, non-delay schedule. The bottom schedule delays the end of job 1 and therefore left the machine idle, while we could finish job 1.

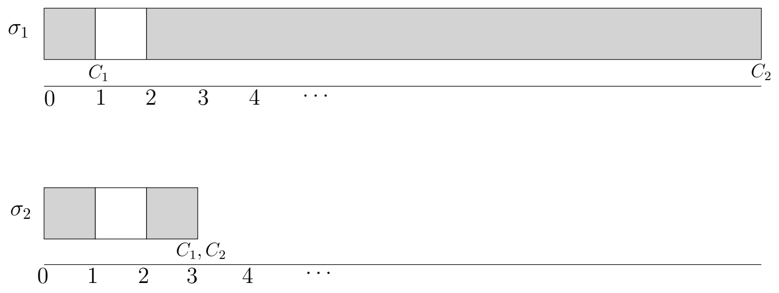

Figure 7.2: Top: Unique non-delay schedule. Bottom: schedule with unforced idleness.

We look at the objective functions for both schedules:

$$\sum C_j^{\sigma_1} = C_1^{\sigma_1} + C_2^{\sigma_1} = 1 + (A+2) = A + 3$$
$$\sum C_j^{\sigma_2} = C_1^{\sigma_2} + C_2^{\sigma_2} = 2\left(2 + \frac{A}{A}\right) = 6$$

We compare the objective value of the unique non-delay schedule σ_1 to the objective value of σ_2.

$$\frac{OPT_{\text{Non-delay}}}{OPT} \geq \lim_{A\to\infty}\left(\frac{\sum C_j^{\sigma_1}}{\sum C_j^{\sigma_2}}\right) = \lim_{A\to\infty}\left(\frac{A+3}{6}\right) = \infty.$$

Therefore a non-delay schedule can be arbitrarily bad compared to the optimal schedule.

□

Corollary 7.3. *An optimal schedule for JDMS($r_j, w_j = 1$) may require unforced idleness.*

Chapter 8

Results and discussion

In this thesis we defined problem JDMS and looked at several variations. The table below is a summery of the problems we solved:

	Without preemption	With preemption
JDMS($w_j = 1$)	Solvable in polynomial time (Chapter 4).	Solvable in polynomial time (Chapter 5).
JDMS($p_j = 1$)	Solvable in polynomial time (Chapter 4).	Solvable in polynomial time (Chapter 6).
JDMS	Remains open	Remains open
JDMS($r_j, w_j = 1$)	Strongly NP-hard, as it is already strongly NP-hard when machine speed is fixed [16].	Solved in polynomial time by SRPT when $\overline{s_i}$ is non-decreasing (Chapter 7). Remains open for an arbitrary speed function.

We discuss these results briefly.

The first results we obtained were for JDMS without preemption (Chapter 4). For both JDMS($w_j = 1$) and JDMS($p_j = 1$) without preemption we succeeded in constructing an optimal solution by rewriting the objective function as $\sum_{i=1}^{n} a_{(i)} q_i$, where $a_{(i)}$ was either the processing time or weight on position i and q_i some constant. Such a function is minimized by assigning the smallest a_i to the largest q_i and thus in both cases an optimal schedule can be created.

We also looked at JDMS without preemption. In this case we did not find an optimal solution, nor proved that it is weakly or strongly NP-hard. We did prove that the optimal policy when the machine speed is constant, WSPT, can be arbitrarily bad in this case.

The most remarkable results in this thesis were the ones for JDMS($w_j = 1$) and for JDMS($p_j = 1$) (Chapter 5, 6). We developed an algorithm that, if we request some order on the completion times, will construct an optimal schedule for that specific order. For problems JDMS($w_j = 1$) and JDMS($p_j = 1$) we succeeded in finding these orders:

- For the problems JDMS($w_j = 1$) we proved that when $p_1 \leq \cdots \leq p_n$, then there will always be an optimal solution such that $C_1 \leq \cdots \leq C_n$, and thus by requesting the order $C_1 \leq \cdots \leq C_n$ Algorithm 5.6 will find an optimal solution for JDMS($w_j = 1$).
- For JDMS($p_j = 1$) we proved that when $w_1 \geq \cdots \geq w_n$, there will always be an optimal solution with $C_1 \leq \cdots \leq C_n$, and thus by requesting the order $C_1 \leq \cdots \leq C_n$ Algorithm 6.2 will find an optimal solution for JDMS($p_j = 1$).

For JDMS we have not found an order such that there is always an optimal solution that obeys this order. When the machine speed is constant Smith [22] proved that WSPT results in an optimal schedule. For JDMS the WSPT order can be arbitrarily bad.

In Chapter 7 we had a quick look on JDMS($r_j, w_j = 1$). We proved that a non-delay schedule may be arbitrarily bad, thus sometimes unforced idleness is necessary to construct an optimal schedule. If the speed function is non-decreasing, JDMS($r_j, w_j = 1$) can be solved in polynomial time by SRPT. For an increasing or arbitrary speed function the problem remains unsolved.

8.1 Open problems

Not all problems that we mentioned are solved. There are three problems that are interesting for further research:

1. JDMS without preemption.
   We proved that WSPT can be arbitrarily bad.
2. JDMS.
   We proved that WSPT can be arbitrarily bad when the speed is increasing. We have also given an example that WSPT is not necessarily optimal when the speed is decreasing. Though it is not clear if WSPT can be arbitrarily bad in this case. Algorithm 6.2 can find the optimal solution for any requested order on completion times. As the number of orders on completion times is of order
3. JDMS($r_j, w_j = 1$) with arbitrary or non-increasing speed function.
   SRPT is optimal when $\overline{s_i}$ is non-decreasing, but when $\overline{s_i}$ is arbitrary an optimal solution may require unforced idleness. A non-delay schedule can be arbitrarily bad.

Bibliography

[1] S. Albers *Review articles: Energy-Efficient Algorithms* In: Communications of the ACM, Vol. 53 No. 5, Pages 86-96, 2010.

[2] S. Albers, H. Fujiwara. *Energy-efficient algorithms for flow time minimization.* In ACM Trans. Alg. 3(4), 49, 2007

[3] S. Albers, O. J. Boxma, K. Pruhs, *Scheduling (Dagstuhl Seminar 13111)*, In: Dagstuhl Reports, 2013, volume 3, number 3, pp. 29.

[4] A. Antoniadas, C. Huang *Non-preemptive speed scaling* In: Fomin, F. V., LNCS, vol 7357, pp. 249-260. Springer, Heidelberg, 2012.

[5] P. Brucker, S. Knust *Complexity results for scheduling problems*, Universitaet Osnabrueck. http://www.informatik.uni-osnabrueck.de/knust/class/

[6] W.L. Eastman, S. Evens, I.M. Isaacs, *Bounds for the optimal scheduling of n jobs on m processors.* In: Management Sci. 11(2), 268-279, 1964.

[7] L. Epstein, A. Levin, A. Marchetti-Spaccamela, N. Megow, J. Mestre, M. Skutella, L. Stougie. *Universal sequencing on a single machine.* In: LNCS Volume 6080, pp 230-243. Springer, Heidelberg 2010.

[8] H.L. Gantt, *Work, Wages and Profit.* In: The Engineering Magazine. New York, 1910; republished as Work, Wages and Profits, Easton, Pennsylvania, Hive Publishing Company, 1974.

[9] S. Gawiejnowicz, *A note on scheduling on a single processor with speed dependent on a number of executed jobs.* In: Information Processing Letters 57, p 297-300, 1996.

[10] R.L. Graham, E.L. Lawler, J.K. Lenstra, A.H.G. Rinnooy Kan, *Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey.* In: Proceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium. Elsevier. pp. (5) 287326, 1979.

[11] D. Grunwald, P. Levis, C.B. Morrey, M. Neufeld. *Policies for dynamic clock scheduling.* In: Proc. OSDI, pp. 73-86, 2000

[12] W. Höhn, T. Jacobs, *On the performance of Smith's rule in single-machine scheduling with non-lineair cost.* In: Fernández-Baca, D. (ed) LATIN 2012. LNCS, vol. 7256, pp. 482-493. Springer, Heidelberg 2012.

[13] S. Iranyi and K.R. Pruhs, *Algoritmic Problems in Power Management.* In: SIGACT News, vol 36, no 2, p 63-76, 2005.

[14] J. Labetoulle, E.L. Lawler, J.K. Lenstra, and A.H.G. Rinooy Kan. *Preemptive scheduling of uniform machines subject to release dates.* In: W.R. Pulleyblank, editor, Progress in Combinatorial Optimization, pages 245,261. Academic Press, 1984.

[15] T.W. Lam, L.K. Lee, I.K.K. To, P. W. H. Wong. *Speed Scaling Functions for Flow Time Scheduling based on Active Job Count* In: LNCS, Volume 5193, pp. 647-659, Springer 2008.

[16] J.K. Lenstra, A.H.G Rinnooy Kan and P. Brucker. *Complexity results of machine scheduling* In: Annuals of Discrete Mathematics, 1, pp. 343-362, 1977.

[17] N. Megow and J. Verschae *Dual Techniques for Scheduling on a Machine with Varying Speed.* In: ICALP 2013, Part 1, LNCS 7965, pp 745-756, 2013.

[18] G. E. Moore. *Cramming more components onto integrated circuits.* In: Electronics Magazine, p 4, 1965

[19] C. Papadimitriou, *Computational Complexity*, Addison Wesley, Reading, MA, 1994.

[20] M.L. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 4th edition, Springer, 2012.

[21] L. Schrage, *A Proof of the Optimality of the Shortest Remaining Processing Time Disciplne,.* In: Operations Research, Vol 16, No 3, pp. 687-690, 1986.

[22] W. E. Smith, *Various optimizers for single-stage production* In: Naval Research Logistics Quarterly, 3:5966, 1956

[23] S. Stiller, A. Wiese, *Increasing speed scheduling and flow scheduling.* Un: ISAAC 2010, Part II. LNCS, vol 6507, pp. 279-290. Springer, Heidelberg 2010.

[24] G. Wang, H. Sun, C. Chu, *Preemptive scheduling with availability constraints to minimize the total weighted completion times.* In: Annals of Operations Research. 133, p 183-192, 2005

[25] A. Wierman, L.L.H. Andrew, and M. Lin. *Handbook on Energy-Aware and Green Computing*, In: Chapter Speed Scaling: An Algorithmic Perspective, pages 385-406. CRC Press, 2012.

[26] F.F. Yao, A.J. Demers and S. Shenker. *A scheduling model for reduced CPU energy* In : FOCS 1995, 374-382.