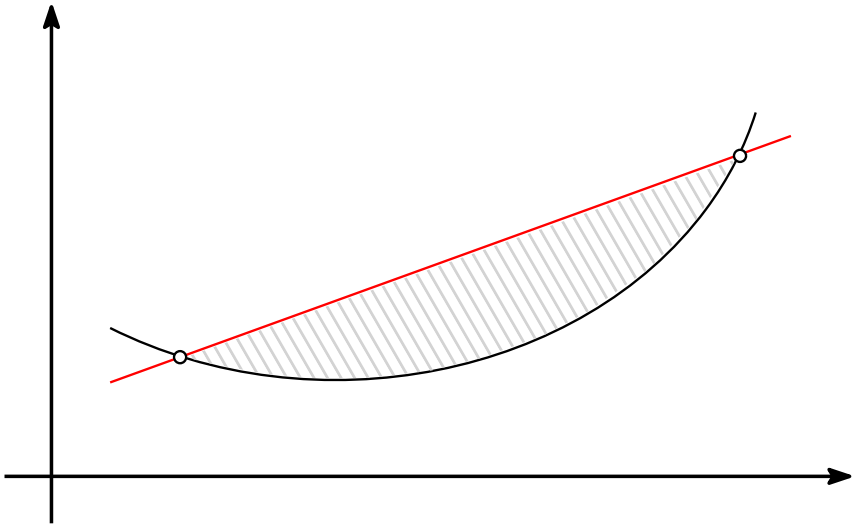
CHALMERS



Improvements in the Non-Preemptive Speed Scaling Setting

Master's Thesis in Computer Science – Algorithms, Languages and Logic

PEDRO MATIAS

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden

Improvements in the Non-Preemptive Speed Scaling Setting
PEDRO MATIAS

Examiner: DEVDATT DUBHASHI

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden October 2015

Abstract

The *speed scaling* problem was first introduced by Yao, Demers and Shenker [35]. It consists on finding a schedule of jobs that minimises the amount of energy that is necessary to execute them on a single *variable-speed* processor. Energy consumption is directly given by a convex function of the processor's speed and each job must be fully executed within its lifetime, which is specified by its work volume, release time and deadline. In the original version of the problem, which is in P, the processor is preemptive. This setting has already been analysed to a great extent, including for multiple processors. Unfortunately, the non-preemptive version of the problem is strongly NP-hard and not so much is known in this variant. Hence, the present work doesn't consider preemption.

The contributions of this thesis can be grouped into two parts. The main results of the first part (chapter 3) include (using a single processor): (i) a substantial improvement in the time complexity when all jobs have the same work volume and (ii) a proof that, when the number of jobs with unrestricted work volume is limited to a constant, the problem is still in P. The second part (chapter 4) presents and proves the correctness of an algorithm that solves a special instance of a different problem: *scheduling with job assignment restrictions* (first introduced by Muratore, Schwarz and Woeginger [29]). The goal is to find a schedule of jobs that minimises the maximum job completion time, over a set of single-speed processors. Solving this problem might give some insight on how to solve the non-preemptive speed scaling problem.

Keywords. Energy-efficiency, scheduling, parallel machines, optimization, speed scaling, algorithm correctness.

Contents

List of Algorithms

List of Theorems

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays, the human being consumes, constantly, a huge amount of energy, throughout the world. According to the International Energy Agency, the total energy used by all humanity in 2012 was, approximately, 13371Mtoe[(1)] [1]. This is equivalent to spending, in average, 4.9×10^6J of energy, every second, 24 hours a day, during one whole year! Adding to this, is the fact that the world energy consumption continues to increase year after year (see fig. 1.1). Perhaps, this boost is, partially, explained by the constant need of humankind to develop new and better technologies that are able to improve its life quality, by promoting a better understanding of the human race, of its needs and of the environment it lives in. However, it is crucial that such development be carried in a sustainable way, so that Earth's ecosystem remains stable and, hence, able to sustain life. Our lives quality depend, intrinsically, on the life quality of the planet we live in.

The present work is concerned with minimising the energy consumption of processors. As it is well known, these devices play a key role in today's urge for technology innovation and development. Moreover, they offer a wide variety of countless applications, ranging from tiny microchips to very large data centers. According to the famous Moore's Law, computing power doubles every 2 years, approximately. Although not necessarily true, it is reasonable to think that one of the implications of this law is that energy consumption increases as well: even if more power doesn't directly imply a boost in energy consumption, it definitely decreases the time needed to accomplish the same tasks and, therefore, creates more room for processing other energy dependent tasks. Furthermore, an increase in power consumption implies an increase in the costs of electricity and heat dissipation. It is important to keep systems running at an appropriate temperature, to allow hardware components to keep on running with the same performance and reliability.

More specifically, this thesis is concerned with finding ecological solutions to the problem of scheduling jobs in processors of *variable-speed*. As the name suggests, variable-speed processors can work at different speeds and, naturally, they consume less energy

[(1)] Mtoe corresponds to the amount of energy produced by burning one million tonnes of crude oil [1]

when working at lower speed. An ecological and feasible solution is one that ensures that all jobs are fully executed, while keeping the amount of energy used to minimum. A more detailed explanation is given in the next section.

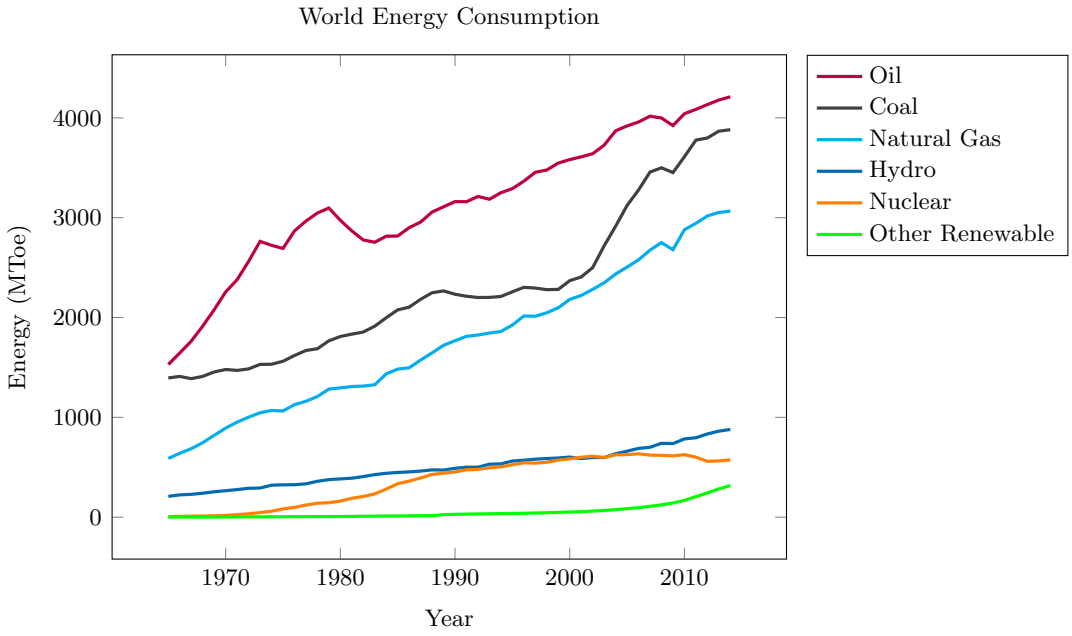


Figure 1.1: Evolution of the consumption of different sources of energy over the years [31].

1.1 Notation and preliminaries

The *Speed Scaling* problem was first introduced by Yao, Demers and Shenker in [35]. The term "speed scaling" derives from the fact that processors can vary its speed even during the execution of a job. The problem formulation is the following. Given a set J of jobs, each with a release time r_j, a deadline d_j and a work volume w_j (for some job $j \in J$), find a schedule of all jobs in J that minimises the amount of energy necessary to execute them on a single variable-speed processor. The energy consumed during an interval of time $[a, b)$ is given by the following integral:

$$\int_a^b s(t)^\alpha dt,$$

In the above expression, $s(t)^\alpha$ denotes the power consumption of the processor (for some constant $\alpha > 1$[(1)]), when it's working at a speed of $s(t)$. Thus, if the processor executes a job j at a constant speed s it will take w_j/s time units to complete it. It is useful to think of the work volume of each job as the number of CPU cycles that are required to execute it.

In the original version of the problem, preemption[(2)] is allowed and every job must be

[(1)] On most modern microprocessors, α is valued between two and three ([14, 34]).
[(2)] See section 1.3 for an explanation of this term.

executed to completion within its *lifetime*. This implies processing all its work volume w_j within the time interval $[r_j, d_j)$. If this isn't the case for every job in J, the schedule is not feasible. Note that there always exists a feasible schedule, since we consider that there are no restrictions on the processor speed, i.e., it can be infinitely fast.

1.2 Goal and overview

Until recently, only the preemptive version of the speed scaling problem was considered. In this thesis, however, we will only be concerned with the non-preemptive case, since there is very little scientific knowledge on this variant of the problem. Initially, the main goal of this thesis was to improve the (so far) best approximation ratio[(1)] in the case that all jobs have the same work volume and there are multiple processors. The idea was to only rely on the design of combinatorial algorithms, as opposed to adopting linear programming (LP) solutions (which appears to be the common strategy in the current literature – see chapter 2). The reasons for adopting an algorithmic point of view are: on one hand, the lack of combinatorial results (in the time of the writing of this thesis) and, on the other hand, the fact that LP methods are, usually, much slower than combinatorial ones and they don't really capture the essence of the problem. Although it requires some insight to be able to formulate the problem as a linear program, LP resembles a "black-box" that can be used to solve any kind of convex optimisation problems.

Unfortunately, it wasn't possible to achieve this task within the desired time window, given its difficulty. Hence, the goal changed into solving, in polynomial-time, special instances of the non-preemptive speed scaling problem. The instances that were studied were combinations of the following variants:

(i) dealing with more than one machine;

(ii) limiting the number of jobs with unrestricted size to a small constant;

(iii) restricting jobs lifetimes according to special rules (see section 1.4).

Sadly, only some of the tackled special cases resulted in reasonable scientific progress (see chapter 3). The study and analysis of some of these special instances opened up the possibility for (i) improving the time complexity in the case that all jobs have the same work volume (see section 3.1), or (ii) proving that when a constant – O(1) – number of jobs have no restrictions on their volume, the problem can still be solved in polynomial-time.

In the attempt of looking at this problem from a different point of view (and after being stuck for a while with no major results), a different scheduling problem was considered. Naturally, both scheduling problems are, somehow, related, but a more detailed

[(1)] See section 1.3 for an explanation of this term.

explanation is given in chapter 4. Despite having obtained interesting results, none of them was further employed in the original problem, due to time constraints.

1.3 Basic notions and further notes

This section explains some of the notions present throughout this thesis and it revisits some important concepts concerning near-optimal solutions.

Preemption. When *preemption* is allowed all the processors considered in a given problem are able to interrupt and, later on, restart the execution of a job. Naturally, a *non-preemptive* processor is not allowed to so and must, therefore, finish the execution of one job before it can, either handle other jobs, or go *idle* (i.e., stop its execution). In the presence of more than one processor, we might consider executing a job in several processors. This procedure is called *migration*. In a migratory environment the processors must also be preemptive, but the converse isn't true.

Approximation algorithms. Today, it is common to give near-optimal solutions to problems that are in NP, that is, problems for which there aren't polynomial-time solutions (unless P=NP). As the name indicates, near-optimal algorithms produce solutions (to optimisation problems), whose cost (or objective value) is not too far from the optimal one. One of the ways to prove the quality of near-optimal solutions is by determining its *approximation ratio*, i.e., a factor β of the best objective value:

$$\beta = \frac{\text{OPT}}{\text{ALG}} \ ,$$

where OPT and ALG represent, respectively, the objective values of the optimal solution and the one given by the approximation algorithm. Naturally, the closer β is to 1, the better is the approximation algorithm. This applies for both minimisation and maximisation problems. A solution is a β-approximation if its approximation ratio is β.

A *polynomial-time approximation scheme* (PTAS) is an approximation algorithm that can be tuned up to improve its approximation ratio, according to an additional parameter $\epsilon > 0$. The price is, naturally, the increase of the running time, but the optimal value can be approximated to any desired degree. For minimisation problems, a PTAS produces $(1+\epsilon)$-approximations, whilst for maximisation problems, it builds $(1-\epsilon)$-approximations. The algorithm must run in polynomial-time with respect to the size of the problem for every fixed ϵ. In other words, its running time is independent of ϵ. If the algorithm runs in time polynomial in both the problem size and $1/\epsilon$, then it is a *fully polynomial-time*

approximation scheme (FPTAS).

Structure of the thesis. This thesis is organised as follows. Chapter 2 summarises the state-of-the-art in speed scaling research for both multiple and single processors. Chapter 3 presents the direct contributions made to special instances of the non-preemptive speed scaling problem. Chapter 4 introduces a different problem whose aim is to schedule all jobs in single speed processors while reducing the maximum job completion time (under some job assignment restrictions). Solving this problem might give back some insight on how to optimally schedule jobs in the speed scaling setting. This chapter presents a solution and its proof of correctness for a special case of the problem. Finally, chapter 5 wraps up all the contributions made in this thesis and briefly mentions the directions that can be explored as a continuation of this work.

Note. The size of the speed scaling problem is denoted by $n = |J|$ and, in the presence of multiple processors, m denotes the total number of processors. Sometimes, problems are denoted by an extension to the standard 3-field classification introduced by Graham et al. [21]. For instance, the non-preemptive speed scaling problem is denoted by $S \mid r_j, d_j \mid E$.

1.4 Special instances

Throughout the thesis we consider two special instances to the problem and both of them are related to the structure given by the lifetime of every job. One of them, we denote by *agreeable deadlines* and the other one by *laminar instance*. Both instances are of special scientific interest, because their structures "complement" each other and, therefore, cover a big fraction of the set of all possible instances of the problem[(1)]. *Laminar instances* are specially interesting, because of its extensive applicability (see section 1.4.2) and the fact that they stress how hard it is to solve the non-preemtive speed scaling problem, comparing to the preemptive version. As a matter of fact, one can use optimal preemptive (and polynomial-time) algorithms to solve any *agreeable deadlines* instance of the non-preemptive variant (see section 2.1).

1.4.1 Agreeable deadlines

This is the case when jobs released earlier have earlier deadlines. More precisely, an instance with *agreeable deadlines* is an instance in which: for any two jobs j and j' in J,

[(1)]Naturally, it doesn't cover all possible set of instances given that there can still be cases whose structure is a mix of both special instances.

if $r_j < r_{j'}$, then $d_i \leq d_{j'}$. Figure 1.2 contains an example of such an instance.

This special structure of the jobs lifetimes is very important, because it, somehow, gives hints to the order of execution of jobs, making the problem slightly easier to solve. In fact, as it is mentioned in section 2.1, there is a polynomial-time algorithm ([35]) that always returns a non-preemptive solution to this particular problem. Hence, this special instance is in P.

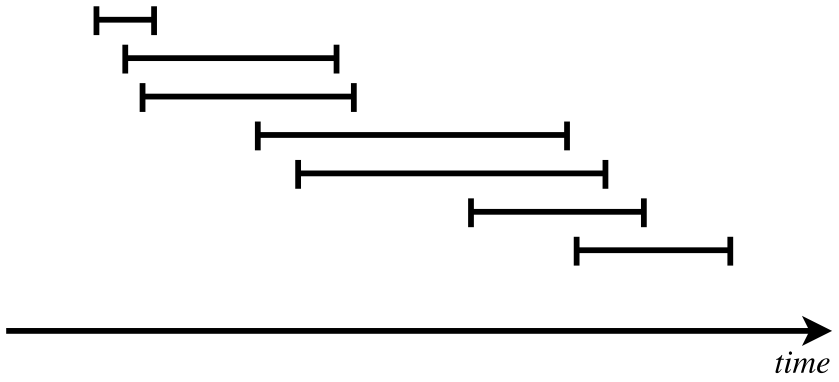


Figure 1.2: Illustration of an instance with agreeable deadlines, where each horizontal segment corresponds to the lifetime of a job.

1.4.2 Laminar instance

An instance is *laminar* if for any two jobs $j, j' \in J$, one of the following happens:

- $[r_j, d_j) \subseteq [r_{j'}, d_{j'})$
- $[r_j, d_j) \supseteq [r_{j'}, d_{j'})$
- $[r_j, d_j) \cap [r_{j'}, d_{j'}) = \varnothing$

This definition can be seen as the "complement" of an instance of agreeable deadlines: for any two jobs with overlapping lifetimes, the one released earlier has a latter deadline.

This kind of problem instances was first explored by Li, Liu and Yao in [28] and it is of scientific interest, because it reflects properties of recursion (see fig. 1.3). Given that there is a wide range of recursive applications in the real world, laminar instances can be used in many practical situations, such as recursive programs[(1)].

In the following section, we consider an even more strict instance to the problem.

[(1)] This is the case where jobs correspond to recursive calls.

1.4.2.1 Purely laminar instance

A special case of laminar instances is when for any two different jobs $j, j' \in J$, we have that:

$$[r_j, d_j) \cap [r_{j'}, d_{j'}) \neq \varnothing.$$

This implies that the lifetimes of all jobs in J form a pile of successive wider lifetimes, if ordered by inclusion (see fig. 1.3). This instance is called *purely laminar* and it is of particular interest because it is easier to solve and its structure is very simple. In fact, there is already a fully polynomial-time approximation scheme[(1)] (FPTAS) for it, which implies that this special instance of the problem is weakly NP-hard [22].

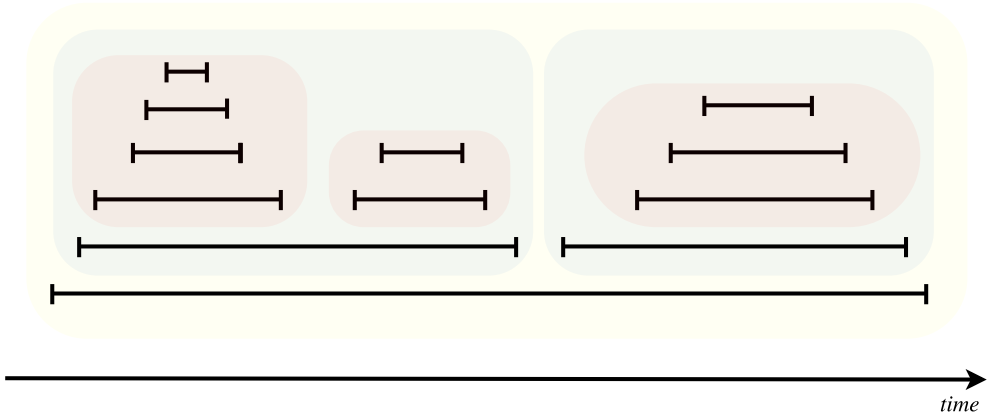


Figure 1.3: Illustration of a laminar instance, where each horizontal segment corresponds to the lifetime of a job. It is easy to see how purely laminar instances (red inner most containers) are a special case of laminar instances.

[(1)] See section 1.3 for an explanation of this term.

Chapter 2

Previous work

This chapter provides an overview of the most relevant theoretical results accomplished in the speed scaling setting, using one or more than one processors.

As it is known, the speed scaling problem was first introduced by Yao, Demers and Shenker [35] in 1995. However, the idea of reducing energy consumption in job scheduling was first studied by Weiser et al. [33] and Govil et al. [20].

2.1 Single processor

Yao, Demers and Shenker gave, in the same paper that formalises the speed scaling problem [35], a very simple greedy algorithm that finds an optimal schedule in polynomial-time. This algorithm was later baptised as YDS, give the names initials. In the same paper, the authors proposed also two online algorithms[(1)] – *Optimal Available* and *Average Rate* – and, for the second one, they proved that its competitiveness[(2)] is $\alpha^\alpha 2^{\alpha-1}$. The first one was analysed by Bansal et al. [12] and they showed that it can achieve a competitive ratio of α^α. Moreover, they improved the competitiveness to $2(\alpha/(\alpha-1))^\alpha e^\alpha$ by designing a new online algorithm. They also showed that the competitive ratio of any randomised algorithm[(3)] is lower bounded by $\Omega((4/3)^\alpha)$.

Until recently, not so much was known in the non-preemptive version of the problem. Its complexity was only established in 2012 by Antoniadis and Huang [6], through a reduction from the 3-Partition problem. In the same paper, the authors gave a $2^{5\alpha-4}$-approximation algorithm for this problem. Later on, this result was improved to $2^{\alpha-1}(1+\epsilon)^\alpha \tilde{B}_\alpha$ (for any $\alpha < 114$) in [10] where $\tilde{B}_\alpha$ is the α-generalized version of the Bell number,

[(1)] An online algorithm doesn't know the whole input data in advance and must, therefore, complete each subsequent request with some level of uncertainty.

[(2)] The competitiveness of an online algorithm compares its performance against an optimal *offline* algorithm that is able to see "future" input [30]. More specifically, it's defined as the worst-case ratio between the objective values of the solutions given by the online and optimal offline algorithms.

[(3)] This is a kind of algorithm that uses probabilistic methods as part of its policy.

introduced by the authors. After that, Cohen-Addad et al. improved the bound to $(12(1+\epsilon))^{\alpha-1}$ [15], for any $\alpha > 25$. In [11], the authors explored the idea of transforming an optimal preemptive schedule to a non-preemptive one. In addition, they achieved an approximation ratio of $(1+\frac{w_{max}}{w_{min}})^{\alpha}$, by using the optimal preemptive solution as a lower bound. In the special case of equal work volumes, this ratio becomes constant: 2^{α}. The current best approximation ratio for this setting is $(1+\epsilon)^{\alpha}\tilde{B}_{\alpha}$ for all $\alpha < 77$, as described in [9]. For the case that jobs share the same work volume, optimal polynomial-time algorithms (based on dynamic programming) were given (independently) by both Angel et al. [4] and Huang and Ott [22]. Other special cases of the problem (concerning the structure of the life intervals of jobs) were also taken into account in [22]. For *agreeable* instances (see subsection 1.4.1), the solution given by Yao et al. in [35] is optimal since it never builds a preemptive schedule [6].

Another way of reducing power consumption is by admitting that the processors can be idle and, therefore, be in a state that uses, substantially, less energy. This mechanism, generally known as *Sleep State*, considers also the fact that awaking processors from their sleep state requires some amount of extra energy. The idea of considering this energy reduction mechanism in the speed scaling setting was first explored by Irani, Shukla and Gupta in [24], where it is given an offline algorithm with an approximation ratio of 3. More recently, Antoniadis, Huang and Ott [7] designed an FPTAS for the same problem.

2.2 Multi-processor

In this section, we distinguish between *homogeneous* and *heterogeneous* processors. Processors are homogeneous if their power consumption is equivalent, or otherwise, they are considered heterogeneous [9]. Moreover, in a *fully heterogeneous* environment, the work volume, release date and deadline of each job is dependent on the processors that executes it.

In the multi-processor environment there are several contributions for the preemptive and (non)migratory settings, but not so much for the non-preemptive version, whose current solutions rely on linear programming (LP) configurations. For the cases when both preemption and migration are allowed there are a few polynomial-time algorithms in the literature, namely in [2, 5, 8, 13]. In the fully *heterogeneous* version, Bampis et al. [10] gave a FPTAS, which relies on an LP formulation. The problem becomes strongly NP-hard when preemption is allowed but not migration [3]. In this article it's also described how a PTAS[(1)] can be achieved on the special case of equal release dates and deadlines for all the jobs. Without this last restriction and, when the set of processors is fully heterogeneous, there exists an approximation algorithm (again, based on LP) of ratio $(1+\epsilon)^{\alpha}\tilde{B}_{\alpha}$ [10].

[(1)] See section 1.3 for an explanation of this term.

Regarding non-preemption and homogeneous processors, Cohen et al. [15] achieved an approximation ratio independent of the number of processors: $(5/2)^{\alpha-1}\tilde{B}_\alpha((1+\epsilon)(1+\frac{w_{max}}{w_{min}}))^\alpha$. This result was recently improved in [9] to $\tilde{B}_\alpha((1+\epsilon)(1+\frac{w_{max}}{w_{min}}))^\alpha$ and it is also valid for the fully heterogeneous setting.

Chapter 3

Analysis and improvements

The non-preemptive speed scaling problem with one processor is strongly NP-hard. This was demonstrated by Antoniadis and Huang in [6] with a simple reduction from the 3-Partition problem. Nevertheless, when all jobs have the same work volume, the problem becomes polynomial-time solvable. Two independent and exact solutions were given (Huang and Ott [22] and Angel et al. [4]) and both of them rely on a dynamic program that takes $O(n^{21})$ time to be solved. In section 3.1 it is shown, by a simple analysis, how this time complexity can be improved by a factor of n^8. Furthermore, section 3.2 demonstrates that, when only a constant number of jobs have unrestricted volumes, the problem is still in P. After these results, more general instances of the problem were explored, such as: increasing the number of different job sizes, or assuming that more than one processor can schedule jobs. Unfortunately, solving these turned out to be quite hard and only results of small impact were obtained (see section 3.3). The main obstacle to achieving reasonable scientific progress was (every time) handling nested lifetimes, i.e., those that define laminar instances (see section 1.4.2). This highlights even more the difficulty of solving such instances of the problem. Recall that, in the presence of agreeable deadlines, the non-preemptive speed scaling problem is in P (as mentioned in section 2.1). Moreover, the proof of NP-hardness for the general non-preemptive speed scaling problem uses, by itself, a laminar instance of the problem (see [6]).

3.1 Jobs of equal work volume

We consider a variation of the non-preemptive speed scaling problem where all jobs have the same work volume, i.e., $w_j = w$ for all $j \in J$. We denote the problem by $S \mid r_j, d_j, w_j = w \mid E$, an extension to the standard 3-field scheduling notation introduced in [21].

We start by briefly explaining the crucial argument for the optimality of the polynomial-time solutions given in Huang and Ott [22] and Angel et al. [4].

Let us call *event* a time point t such that: $t = r_j \vee t = d_j$ for some job $j \in J$.

Observation 1. *In an optimal schedule, the processor only changes speed at events.*

Observation 1 follows from Jensen's Inequality[(1)] [25] and the convexity of the power function associated with the processor. A detailed proof is given by Angel et al. in [4, Proposition 5].

Both papers [22] and [4] take advantage of obs. 1 to define a polynomial-size set (w.r.t. $|J|$) whose elements are the endpoints of possible optimal execution intervals for a job.

From now on, we will focus on the solution given by Huang and Ott [22], because it's simpler. In their paper, the set mentioned above is denoted by $\mathcal{P}_{exact}$ and its elements are called *grid points*.

Definition 1 ($\mathcal{P}_{exact}$)**.** For every two events t, t' and for every $k \in \{1, \ldots, n\}$, create $k-1$ equally spaced grid points that partition $[t, t')$ into k subintervals of equal length. Moreover, create grid points for t and t'. The set $\mathcal{P}_{exact}$ is defined by the union of all these grid points.

As a consequence of obs. 1, the authors make the following deduction:

Observation 2. *In all optimal schedules, every job is executed in an interval $[s, t)$, where s and t are elements of $\mathcal{P}_{exact}$.*

Before proving obs. 2, we need to introduce the concept of *phase*, originally used in [22, Proof of Lemma 2].

Definition 2 (Phase)**.** A *phase* is a time interval in which the processor works at a constant speed, for a given schedule of jobs.

Proof of obs. 2. We reproduce the proof given in [22, Proof of Lemma 2]. Let S^* be any optimal schedule. We know from obs. 1 that the endpoints of every phase correspond to events. Moreover, every job in S^* is processed using an uniform speed. This follows directly from Jensen's inequality. Hence, every phase defines an interval where a number k of jobs are executed to completion, in a uniform speed. What's more, their execution intervals have the same length in that phase, since all jobs have the same work volume.

[(1)] An illustration of Jensen's Inequality is given in the front page of this thesis.

In other words, every phase in S^* is defined by consecutive execution intervals of the same length. This implies a partition of each phase by $k-1$ equally spaced time points. It is clear from its definition, that $\mathcal{P}_{exact}$ includes all these partition points for every possible phase and for every $k \in \{1, \ldots, n\}$. □

The next section specifies how the authors solve this problem, using the above information.

3.1.1 The dynamic program

Following is an explanation of how Huand and Ott [22] solves the problem using dynamic programming. For a more detailed explanation the reader is recommended to look at the original paper.

Every dynamic program (DP) must satisfy 2 properties: *optimal sub-structure* and *overlapping sub-problems*.

Optimal sub-structure. One of the consequences of obs. 2 is that we can try all possible schedules for each job $j \in J$ in polynomial-time, according to the set $\mathcal{P}_{exact}$. For each attempt, we then focus on two different sub-problems: one schedules jobs before j and, the other one, after. Figure 3.1 illustrates the idea.

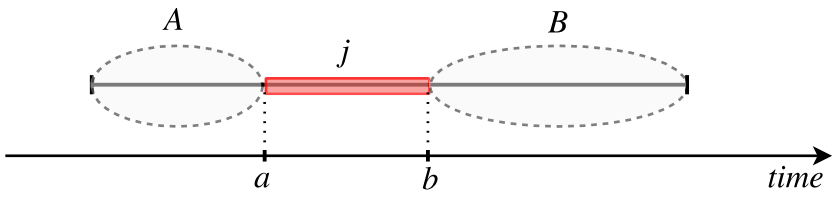


Figure 3.1: Recursive step in the dynamic program divides main problem into two sub-problems (A and B), according to a partition of the time axis. The red rectangle corresponds to the execution interval $[a, b)$ of job j and both a and b belong to $\mathcal{P}_{exact}$.

If these two sub-problems are independent, then an optimal solution to both combined with the optimal execution interval of j gives an optimal solution to the main problem. The difficulty relies on ensuring that the sub-problems are independent, which, in this case, implies partitioning the remaining jobs into two sets. Huang and Ott solved this issue by introducing the notion of a *lexicographic order* of schedules. Considering this, jobs are partitioned according to the order of jobs given by the smallest lexicographic schedule that processes j in its guessed execution interval. More specifically, if J_q is to be executed in the interval $[a, b)$, then all jobs $\{J_k \mid k > q \ \wedge \ d_{J_k} \leq e_{J_q}\}$ must be completely processed before J_q, given the deadline constraints. Moreover, all jobs $\{J_k \mid k > q \wedge e_{J_q} < d_{J_k}\}$ must be completely processed after J_q, or otherwise the resulting schedule isn't lexicographically

smallest (easily verified by contradiction, using a simple jobs swapping argument)[1].

Overlapping sub-problems. Every problem is divided into independent sub-problems by partitioning the set of remaining jobs according to the time horizon. From this, it is clear how, eventually, sub-problems will overlap (i.e., be solved more than once).

Having covered all properties of a DP, we are now in conditions of (re)formulating the program designed by Huang and Ott.

Definition 3 (Dynamic program according to [22])**.** Let $E(i, g_1, g_2, g_3)$ define an optimal schedule for jobs $\{J_k \mid k \geq i \wedge g_1 < d_{J_k} \leq g_3\}$ using only the interval $[g_1, g_2)$. It is important that no job is scheduled in $[g_2, g_3)$. Jobs in J are considered to be ordered by increasing release times:

$$r_{J_1} \leq r_{J_2} \leq \ldots \leq r_{J_n}.$$

The DP is given by the following recurrence relation:

$$E(i, g_1, g_2, g_3) := \min_{a,b \in \mathcal{P}_{exact}} \{\ \mathit{energy}(q, [a, b)) + \ E(q+1, g_1, a, b) \ + \ E(q+1, b, g_2, g_3) \mid (g_1 \leq a < b \leq g_2) \wedge (a \geq r_q) \wedge (b \leq d_q)\ \},$$

where:

- q is the smallest job index in $\{J_k \mid k \geq i \wedge g_1 < d_{J_k} \leq g_3\}$
- a and b are (respectively), the beginning and end of J_q's execution
- $\mathit{energy}(x, \Delta)$ is the amount of energy required to process job J_x during interval Δ

The optimal schedule is given by $E(1, r^*, d^*, d^*)$, where r^* is the earliest release time and d^* is the latest deadline over all the jobs in J. Base cases are given by:

$$E(i, g_1, g_2, g_3) = \begin{cases} 0, & \text{if} \quad \{J_k \mid k \geq i \wedge g_1 < d_{J_k} \leq g_3\} = \varnothing \\ \infty, & \text{if} \quad \exists k \geq i : g_1 < d_k \geq g_3 \wedge [r_k, d_k] \cap [g_1, g_2) = \varnothing. \end{cases}$$

Complexity. To analyse the complexity, we must know the size of $\mathcal{P}_{exact}$. According definition 1, for every pair of events (t, t') we create:

$$\sum_{k=1}^{n} k + 1 = O(n^2) \quad \text{grid points.}$$

[1] The reader can find more information in [22, section 4]

Furthermore, the total number of pairs of events is $2n \cdot (2n-1) = O(n^2)$ (recall obs. 1). Hence:

$$|\mathcal{P}_{exact}| = O(n^4).$$

The overall time complexity of the algorithm is $O(n^{21})$, since the DP computes $O(n \cdot |\mathcal{P}_{exact}|^3)$ entries and, for each entry, it minimizes over $O(|\mathcal{P}_{exact}|^2)$ different possibilities – one for each possible execution interval $[a, b)$ of a job.

3.1.2 Improving the time complexity

In this section it is shown how we can further improve the time complexity of the solution explained in section 3.1.1, by simply doing a better analysis of the problem. The method described here can, obviously, be adapted to the solution given in [4] by Angel, Bampis and Chau.

We exploit the fact that, in order to determine the execution interval of a job, it is only necessary to look at pairs of grid points that were created (simultaneously) during the partition of a specific interval of time defined by two events (see definition 1 – $\mathcal{P}_{exact}$). With this idea in mind, we can reformulate the problem and replace the notion of grid points with the notion of *grid intervals*.

Definition 4 (Grid intervals set $\mathcal{I}$)**.** For every two events a, b and for every $k \in \{1, \ldots, n\}$, partition $[a, b)$ into k *grid intervals* of equal length. The union of all these intervals defines the set $\mathcal{I}_{a,b}$. Formally,

$$\mathcal{I}_{a,b} = \bigcup_{1 \le k \le n} \{\ [\ a + i\delta,\ a + (i+1)\delta\ )\ \mid 0 \le i \le k-1\},$$

where $\delta = \frac{b-a}{k}$.

Finally, the intervals set is given by

$$\mathcal{I} = \bigcup_{a,b} \{\ \mathcal{I}_{a,b} \mid a, b \text{ are events }\}.$$

Although slightly smaller[(1)], the size of this set is of the same order of complexity as $|\mathcal{P}_{exact}|$, i.e., $|\mathcal{I}| = O(n^4)$. However, one can guess the execution interval of a job by simply looking at the elements of $\mathcal{I}$, instead of $(\mathcal{P}_{exact})^2$.

Lemma 1. *In all optimal schedules, every job is executed in one of the intervals of $\mathcal{I}$.*

[(1)] For each pair of events we create k intervals, instead of $k-1$ points

Proof. We know from the proof of obs. 2 that, in any optimal solution, every job is executed in an interval defined by time points that partition a specific phase (recall definitions 1 ($\mathcal{P}_{exact}$) and 2 (phase)). Considering this, it is clear from its definition, that the set $\mathcal{I}$ only includes such intervals of execution. □

In the DP definition in section 3.1.1, we are taking into consideration execution intervals defined by grid points that might "belong" to different phases. This is unnecessary and inconsistent with the proof of obs. 2, which states that, in any optimal schedule, jobs are executed in a specific phase, along with a specific number of jobs. Figure 3.2 illustrates an example of a unnecessary attempt of scheduling a job.

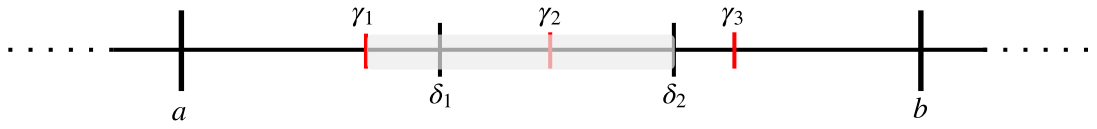


Figure 3.2: Example of an unnecessary attempt of scheduling some job (grey rectangle) – see lemma 1. In this diagram a and b represent the endpoints of a phase (see definition 2) and the timepoints γ_i (resp. δ_i) account for the fact that four (resp. three) jobs are to be scheduled in $[a, b)$.

3.1.2.1 The improved DP

With the new definition of a set of grid intervals, each entry $E(i, g_1, g_2, g_3)$ can be computed by minimising over $|\mathcal{I}|$ different possibilities. One for each possible execution of J_i.

Furthermore, the number of entries can also be reduced. By looking at the recursive definition of $E(i, g_1, g_2, g_3)$ in definition 1, we notice that g_2 and g_3 always correspond (respectively) to the beginning and end of the execution interval of an already scheduled job.

Taking everything into consideration, we can now redefine the dynamic program recurrence relation as follows:

$$E(i, g, \Delta) := \min_{\tau \in \mathcal{I}} \{energy(q, \tau) + E(q+1, g, \tau) + E(q+1, \tau_{end}, \Delta) \mid \tau \subseteq [g, \Delta_{start}) \wedge \tau \subseteq [r_q, d_q)\},$$

assuming that χ_{start} and χ_{end} represent, respectively, the beginning and end time points of an interval χ. Figure 3.3 illustrates the above recurrence relation. The base cases are, similarly, defined.

The partition of jobs for sub-problems $E(q+1, g, \tau)$ and $E(q+1, \tau_{end}, \Delta)$ is done in the same way as in the original DP [22].

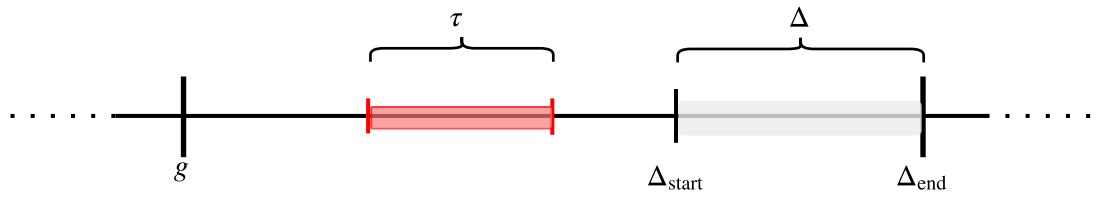


Figure 3.3: Representation of each sub-problem of the improved DP, where τ corresponds to a possible execution interval for the job we are, currently, trying to schedule.

Theorem 1. *The improved DP solves optimally the problem* $S \mid r_j, d_j, w_j = w \mid E$ *in* $O(n^{13})$ *time.*

Proof. Correctness of the improved DP follows easily from lemma 1 and its similarity to the DP defined by Huang and Ott [22]: the only difference between both is the intervals set used. The overall complexity is $O(n^{13})$, since the number of entries is now $O(n \cdot |\mathcal{P}_{exact}| \cdot |\mathcal{I}|)$ and, for each entry, there are $O(|\mathcal{I}|)$ possible ways of executing a job. □

3.2 A constant number of jobs have unrestricted volumes

We consider a variation of the problem defined in section 3.1, by assuming that $c = O(1)$ jobs in J can have any volume. We denote this problem by $S \mid r_j, d_j, w_j \in \{w, w^1, \ldots, w^c\} \mid E$, an extension to the standard 3-field classification introduced by Graham et al. in [21]. In this section, we will show that this problem is still in P.

Let us decompose the set of jobs into $J = J^* \cup J^=$, where J^* is the set of the c jobs that don't have restrictions on their volume and $J^=$ is the set of the the remaining (equal-volume) jobs.

Recall that a processor only changes speed at events (see obs. 1) and that each phase (defined by a pair of events) is an interval where a number of jobs are fully executed, in a uniform speed (see proof of obs. 2). Naturally, we can no longer consider that all jobs (in the same phase) are executed during the same amount of time.

Therefore, we need to extend the definition of the set of grid intervals (see definition 4). We will denote this new set by $\mathcal{I}'$. As in $\mathcal{I}$, this set includes all possible (optimal) intervals of execution for all jobs in J. Notice that, despite its size being increased substantially (we now have c jobs of unrestricted volumes), we will show that it is still in P.

3.2.1 Extended grid intervals set $\mathcal{I}'$

Let K denote a set of jobs that are to be scheduled in a phase $[a, b)$. Moreover, consider a permutation $\pi : K \to \mathbb{N}^+$, which expresses a scheduling order of the jobs in K.

Since the processor speed is constant during the whole phase, we can easily determine the length Δ^j of the execution interval of the jth scheduled job (using the order given by π). From here, we can determine the execution intervals of all jobs in K. Hence, create a grid interval for each of them, excluding the unfeasible intervals.

Now, let us do the same for all permutations π of each set of jobs $K \in \mathcal{P}(J)$, where $\mathcal{P}$ is the power set function. The set of all created grid intervals is denoted by ${\mathcal{I}'}_{a,b}$. Formally,

$${\mathcal{I}'}_{a,b} = \bigcup_{K \in \mathcal{P}(J)} \bigcup_{\pi \in \Pi(K)} \{\ [a, a + \Delta^1), \ldots, [a + \sum_{i=1}^{k-1} \Delta^i, a + \sum_{i=1}^{k} \Delta^i)\ \},$$

where $\Pi(K)$ is the set of all permutations of jobs in K and $\Delta^i = \dfrac{w_l(b-a)}{\sum_{j \in K} w_j}$, such that $\pi(l) = i$.

Finally, let the new set of grid intervals $\mathcal{I}'$ be the reunion of the sets ${\mathcal{I}'}_{i,j}$, for all possible phases:

$$\mathcal{I}' = \bigcup_{i,j} \{\ {\mathcal{I}'}_{i,j} \mid i, j \text{ are events }\}.$$

Lemma 2. *The size of $\mathcal{I}'$ is polynomial, with respect to $|J|$.*

Proof. Since it is hard to give a precise value of the size of $\mathcal{I}'$, we will give an estimation by computing upper and lower bounds.

We begin by estimating the size of ${\mathcal{I}'}_{a,b}$. Assume, for now, that k jobs are to be scheduled in phase $[a, b)$ and that from these, k^* jobs belong to J^* (i.e. have unrestricted volume). The total number of permutations is $k!$, but we are not concerned with the order of jobs of equal volume, since different orderings will always translate into the same set of execution intervals. Hence, the number of relevant permutations to consider is:

$$\frac{k!}{(k-k^*)!} = k(k-1)\ldots(k-k^*+1) < k^{k^*}.$$

For each permutation, we need to also estimate in how many ways we can select k^* jobs from J^*:

$$\binom{c}{k^*} < 2^c = O(1)$$

Finally, by taking everything into consideration, the number of distinct execution

intervals ϕ_{k,k^*} is upper and lower bounded by

$$(k-k^*+1)^{k^*} < \frac{k!}{(k-k^*)!} \leq \quad \boldsymbol{\phi_{k,k^*}} \quad \leq k\frac{k}{(k-k^*)!}\binom{c}{k^*} < k^{k^*+1}2^c \quad .$$

Note that ϕ_{k,k^*} cannot be less than the number of relevant permutations and that, for each of them, we have k intervals. In addition, the value of ϕ_{k,k^*} is upper bounded (and not exact), since there might still be overlapping intervals among different permutations. However, it is hard to also account for these. Thus, we consider only that:

$$\boldsymbol{\phi_{k,k^*}} < k^{k^*+1}2^c = O(k^{k^*+1}).$$

Furthermore,

$$\begin{aligned}
|\boldsymbol{\mathcal{I}'_{a,b}}| &= \sum_{k=1}^{n}\sum_{k^*=1}^{\min(k,c)} \phi_{k,k^*} \\
&\leq \sum_{k=1}^{n}\sum_{k^*=1}^{c} O(k^{k^*+1}) \\
&< c \cdot \sum_{k=1}^{n} O(k^{c+1}) \\
&< c \cdot O(n^{c+2}) \\
&= O(n^{c+2})
\end{aligned}$$

Finally, we can determine the size of the new set of grid intervals. $|\mathcal{I}'| = O(n^2)\cdot|\mathcal{I}'_{a,b}| = O(n^{c+4}) = O(n^{O(c)})$, since there are $O(n^2)$ phases. Since c is a constant, the size of $\mathcal{I}'$ is polynomial with respect to $n = |J|$. □

3.2.2 The solution

The following algorithm is a very naive solution, but it is enough to show that the problem is still in P.

Algorithm Extended DP: $S \mid r_j, d_j, w_j \in \{w, w^1, \ldots, w^c\} \mid E$

1. Compute all possible schedules Ω for jobs J^*

2. FOR EACH schedule ω in Ω DO

- **(a)** Let I be the set of execution intervals of jobs J^* in ω
- **(b)** Let $I' = \{i \mid i \in \mathcal{I}' \text{ and } i \text{ does not intersect any interval in I}\}$
- **(c)** Let s be the schedule that combines ω with the schedule returned by the *improved DP* for the remaining jobs, using the intervals set I'.
- **(d)** IF the amount of energy required in s is the smallest so far THEN
  - **i.** Let $s^* = s$

3. RETURN s^*

The "improved DP" mentioned in step 2.(b) is the one described in section 3.1.2.1.

Theorem 2. *Algorithm Extended DP returns an optimal solution to* $S \mid r_j, d_j, w_j \in \{w, w^1, \ldots, w^c\} \mid E$ *in time polynomial in* $n = |J|$.

Proof. Assuming that c is a small number, we can afford to enumerate all possible schedules of jobs J^* of unrestricted volume. We know that at least one of them is in the optimal solution, so the algorithm finds it. Step b) of the algorithm makes sure that we won't have overlapping execution intervals, which lets us build a feasible and optimal schedule from the solution given by the improved DP (see theorem 1). This way, it follows, from the optimality of the improved DP, that alg. Extended DP returns an optimal solution.

The number of schedules computed in step 1. of the algorithm is $O(|\mathcal{I}'|^c)$. For each of them, the heaviest computation we do is solving the improved DP for a subset of $\mathcal{I}'$, which takes $O(n|\mathcal{I}'|^3)$ time to solve (see lemma 2). Therefore, the overall time complexity is

$$\begin{aligned} O(|\mathcal{I}'|^c) \cdot O(n|\mathcal{I}'|^3) &= O(n|\mathcal{I}'|^{c+3}) \\ &= O(n^{O(c^2)}). \end{aligned}$$

As long as c is a constant, alg. Extended DP builds an optimal schedule in polynomial-time w.r.t. $n = |J|$. □

Corollary 1. *The problem* $S \mid r_j, d_j, w_j \in \{w, w^1, \ldots, w^c\} \mid E$ *is in P.*

3.3 Other improvements and unsuccessful directions

In the attempt of solving the non-preemptive speed scaling problem from different perspectives, other contributions (and unsuccessful attempts) were made to instances of the problem that generalise the case that all jobs have the same volume. Although some of the accomplished improvements are not so relevant, they were included in this section, because they reflect the amount of work and time invested in this thesis. Moreover (and more importantly), the ideas presented here might give insight to more inspiring results in the future. It is worth mentioning that these smaller improvements rely on extensions of the dynamic programming algorithm given by Huang and Ott [22] and revisited here in section 3.1.

After being stuck for a while with no major results in any of the attempted instances of the problem, a different approach was pursued, which consisted in solving a different scheduling problem (see chapter 4).

3.3.1 Single processor

Following is an unsuccessful attempt at extending the solution given by Huang and Ott [22] to include two job volumes, instead of only one. After this, it is shown how we can improve the time complexity of the same algorithm when in the presence of a purely laminar instance. Towards the end of this section, the reader can have an idea of why this simple problem is not as easy to solve as one might think. In addition, some of the (not so lucky) attempts to solve this particular problem are, briefly, mentioned.

Two job volumes. Assume that all jobs have a work volume of either w^1 or w^2 and that w^1 divides w^2. Recall from section 3.1.1 that, when all jobs of equal volume, we partition the jobs in the recursive step of the dynamic program, into two sets $J^- := \{J_k \mid k > q \ \wedge \ d_{J_k} \leq e_{J_q}\}$ and $J^+ := \{J_k \mid k > q \ \wedge \ e_{J_q} < d_{J_k}\}$, that are executed (respectively) before and after J_q (the job we are currently scheduling). Correctness of this partition follows from the lexicographic notion introduced by the authors[(1)]. However, it is easy to see that this strategy no longer works for two job volumes: if $w_{J_q} = w^1$, then the argument is only valid for jobs whose volume is w^1.

For simplicity, assume that $w^1 = 1$ and $w^2 = 2$. Suppose that we are also trying to schedule a job j in the interval $[a, b)$ and that $w_j = 2$. We know that the partition can be made for all jobs of volume 2, so just consider the remaining jobs. From these, it is clear that all jobs j with $d_j \leq b$ or $r_j \geq a$, must go (respectively) to J^- and J^+. The jobs we are left with (let us denote them by $\tilde{J}$) are the problematic ones, since they can be assigned

[(1)] A detailed argument is given in [22, section 4].

to any of the sets. A brute-force solution would try out all $2^{|\tilde{J}|}$ possible combinations of partitioning these jobs. Originally, the idea was to take advantage of the multiplicity of the job volumes and reduce the number of possible combinations by ensuring that no two jobs of $\tilde{J}$ can be scheduled consecutively before J_q. If they were, then we could swap both of them with J_q and get a smaller lexicographic schedule. But, clearly the resulting schedule isn't, necessarily, lexicographically smaller. If this idea worked, it would solve the problem in sub-exponential time (w.r.t. the number of jobs n).

Purely laminar instances. We assume that all jobs have the same work volume. As in section 3.1.1, consider that the jobs are ordered by increasing order of their release times. In the presence of a purely laminar instance, this implies the following:

$$[r_{J_1}, d_{J_1}) \supseteq [r_{J_2}, d_{J_2}) \supseteq \ldots \supseteq [r_{J_n}, d_{J_n})$$

We take advantage of the following observation.

Observation 3. *Let $\pi^S : \{1, \ldots, n\} \to \{1, \ldots, n\}$ denote the order of execution of jobs for a schedule S, such that $\pi^S(i)$ denotes the index of the i^{th} executed job. In a purely laminar setting (where all jobs have the same volume), there exists an optimal schedule S^* of all jobs J such that, for some $k \in \{1, \ldots, n\}$, we have that:*

- $\pi^{S^*}(1) < \pi^{S^*}(2) < \ldots < \pi^{S^*}(k)$, *and*
- $\pi^{S^*}(n) < \pi^{S^*}(n-1) < \ldots < \pi^{S^*}(k)$.

Proof. We want to prove, in other words, that there exists a schedule S^* such that the function π^{S^*} has only one local (and, hence, global) maximum. If not, then there is a job executed in the l^{th} position (for some $l \in \{2, \ldots, n-1\}$) such that:

$$\pi^{S^*}(l-1) > \pi^{S^*}(l) \quad \text{and} \quad \pi^{S^*}(l) < \pi^{S^*}(l+1).$$

Let j_1 and j_2 be the jobs indexed by, respectively, $\pi^{S^*}(l)$ and $\min(\pi^{S^*}(l-1), \pi^{S^*}(l+1))$. Now, let us change S^* by swapping the execution intervals of the jobs j_1 and j_2. It can be easily verified that such modification does not violate any restrictions associated with the release times and deadlines of the involved jobs. If we repeat this procedure until π^{S^*} has only one local maximum, we get a feasible and optimal schedule S^*. □

Intuitively, the above observation tells us that there is always an optimal schedule in which: (i) before the execution of J_n, jobs are processed in increasing order of their indexes and (ii) after the execution of J_n, jobs are processed in decreasing order of their indexes. If J_n is the first (last) job executed, then of course no job is executed before (after) it. Unfortunately, the total number of job orderings is still exponential in n. If i jobs are

to be scheduled before J_n, then there are $\binom{n-1}{i}$ many possible orderings. Hence, for all $0 \leq i \leq n-1$:

$$\sum_{i=0}^{n-1} \binom{n-1}{i} = 2^{n-1} \in O(2^n).$$

Nevertheless, we can still benefit from something: we can, actually, skip the partition of jobs that occurs in the recursive step of the dynamic programming solution (see section 3.1.1). If we try to schedule each job j in J by increasing order of the jobs indexes, we know (according to obs. 3) that j must be executed in one of the two extremes of the only time window that is currently available for scheduling (see fig. 3.4). Thus, we have less ways of possibly scheduling j. Assume w.l.o.g. that j is to be scheduled in the left extreme, right after the processing of job j'. If the end of the execution of j' does not mark the end of a phase[(1)], there is even just one possibility of scheduling j. Otherwise, we have $O(n)$ possible ways of assigning j to a phase that starts immediately after j''s execution, instead of the usual $O(n^2)$ possibilities (see section 3.1.1). Note that the number of possibilities only doubles if we also consider executing j in the right extreme of the available time window. With this idea in mind, a straightforward extension of the previously given dynamic program solves this particular problem in $O(n^8)$ time, improving the time complexity by a factor of n^5. The recurrence relation can be defined as follows:

$$E(i,\Delta) := \min_{\tau \in \mathcal{I}} \{\ energy(i,\tau) + E(i+1, \Delta \setminus \tau) \mid \\ \tau \subseteq \Delta \wedge (\tau_{start} = \Delta_{start} \vee \tau_{end} = \Delta_{end})\ \},$$

where $E(i,\Delta)$ denotes the amount of energy necessary to schedule all jobs $j \in J$ in the available time window Δ, such that $r_j \geq r_i$. Note that, unlike the dynamic program described in section 3.1.2.1, Δ represents the time interval where jobs need to be scheduled. Moreover, we no longer need the extra parameter $g \in \mathcal{P}_{exact}$, which makes a substantial reduction in the time complexity. The base cases can be, trivially, determined from the above recurrence. Unexplained notation is introduced in section 3.1.1.

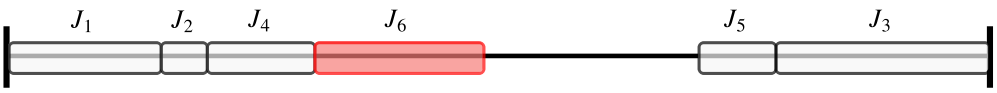


Figure 3.4: One of the consequences of obs. 3 is that we can schedule jobs by increasing indexes to the extremes of the current available time window (assuming jobs are ordered by increasing release times). In this case J_6 denotes the job we are, currently, trying to schedule.

[(1)] a phase is an interval of time (defined by two events) during which the processor doesn't change speed

Note. Unlike the original dynamic programming solution, we don't need to worry about partitioning the set of remaining jobs, when computing each entry of the dynamic programming table: there is only one recursive call and the time window is never split up into additional ones. Nevertheless, it is easy to see that the overlapping sub-structure property† is still present.

†See section 3.1.1 for a more detailed explanation of this concept.

Following is another interesting observation with regard to this special instance of the non-preemptive speed scaling problem.

Observation 4. *In a purely laminar setting (where all jobs have the same work volume), every phase $[a, b)$ of an optimal solution starts by executing a job $j \in J$ if $r_j = a$ and finishes with the execution of some job $j' \in J$ if $d_{j'} = b$.*

Proof. It is easy to see that, in a purely laminar setting where all jobs have the same work volume, every optimal solution has only one phase that starts in some release time r_i and ends in some deadline d_j for two jobs i and j in J. All the phases before the interval $[r_i, d_j)$ are, exclusively, defined by release times and, similarly, all the phases after $[r_i, d_j)$ are, exclusively, defined by deadlines. This follows from the special structure of purely laminar instances (see fig. 3.5).

The rest of the proof follows easily from Jensen's inequality [25]. Suppose that, in an optimal solution, there is a phase $[r_j, y)$ that starts by executing a job j' and such that $r_j \neq r_{j'}$. Then, it must be the case that $[r_{j'}, d_{j'}) \supset [r_j, d_j)$. However, in this case, we could let the phase $[r_j, y)$ start earlier and, therefore, balance out the energy spent during the time given by this and the phase immediately before. Even a tiny change in the starting time is allowed (since $r_{j'} < r_j$) and produces a more efficient schedule, given the convexity of the power function associated with the processor (Jensen's inequality [25]). □

Unfortunately, no benefit was discovered from this last observation.

What makes this simple problem (at first sight) so difficult to solve is the (apparent) non-existence of a greedy strategy that is able to solve the problem of deciding the ordering of jobs. Even by taking into consideration obs. 3, one cannot pre-determine if a given job is supposed to be scheduled either before or after J_n. Other attempts were made to solve this particular issue. One of them (i) consisted in solving a dynamic program that solves sub-problems according to the different job orderings, but no relation could be found between sub-problems having similar job orderings. Another attempt (ii) consisted in identifying

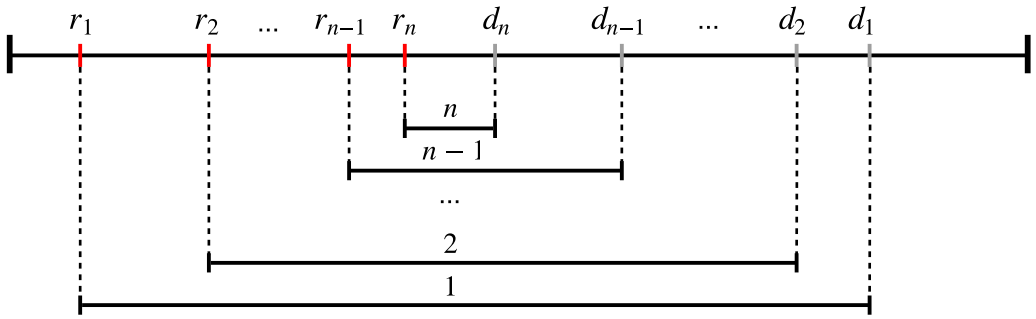


Figure 3.5: Illustration of jobs lifetimes in purely laminar instances. Release dates are given in red and deadlines in grey. Consider that $i = J_i$ (for all i) and that jobs are ordered by increasing release times.

independent sub-problems based on the length of the lifetimes[(1)] of every job. The idea would be to exploit the fact that jobs with very narrow lifetimes will never be executed at the same speed as jobs with substantially bigger lifetimes, if these are in small number. Figure 3.6 illustrates this point. After identifying the simpler sub-problems, one could use the regular DP to solve them independently. Finally, (iii) several reductions to problems involving flow networks were also explored, but none of them yielded interesting results. The general idea was to model the space of possible intervals of execution (in a single processor) as nodes of a flow network. This strategy was based on the work done by: Bampis et al. [8], Albers et al. [2] and Angel et al. [5], in the multi-processor setting and some of the covered problems included variations of the assignment problem [26] and of the minimum cost circulation problem [32]. Unfortunately, the time complexity can easily become huge, given that the number of nodes in the network is at least in the order of $O(n^4)$.

[(1)]Recall that lifetimes are time intervals defined by the release time and deadline of the jobs.

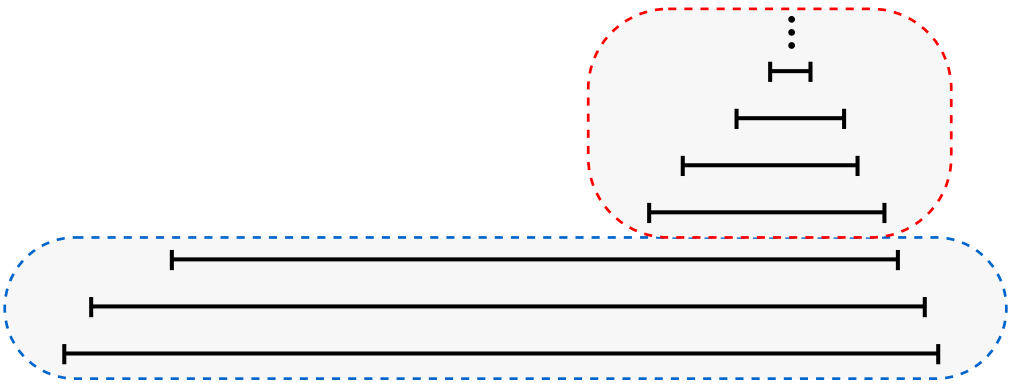


Figure 3.6: Illustration of the idea of exploiting big differences in jobs lifetimes. In this instance, we can solve, independently, each of the sub-problems given by the blue and red containers (each including a set of jobs). It is easy to see that the smallest execution interval in the blue sub-problem will always be greater than the largest execution interval in the red sub-problem.

3.3.2 Multi-processor

Other attempts were also made to solve special instances in the multi-processor non-preemptive setting, but only one had results (see below). Most of the failed experiments consisted in assigning jobs to processors in a greedy fashion, similarly to what has been done by Albers et al. [3] when preemption is allowed and deadlines are agreeable – sort the jobs according to release times, assign them to the processors in a *Round Robin* manner and apply the YDS algorithm on each processor [35].

Purely laminar instances on m processors. Assume, once again, that all jobs have the same work volume. As it is explained in section 3.3.1, one of the consequences of obs. 3 is that we can find an optimal solution in the single processor environment by using the following strategy: schedule each job in increasing order of its release time, to an interval that is immediately before or after the interval of execution of a previously scheduled job. It is easy to see that obs. 3 is still valid in the multi-processor setting (considering each processor, individually). With this idea in mind, we can make a straightforward extension to the dynamic program described in the previous section, so that it works in the multi-processor environment.

We define a dynamic programming algorithm that keeps track of (at most) m available time windows (one for each processor) and schedules all jobs in increasing order of their release dates. The amount of energy necessary to schedule all jobs $j \in J$, such that $r_j \geq r_i$

is computed according to the following expression:

$$E(i, \Delta^1, \ldots, \Delta^m) := \min_{\substack{\tau \in \mathcal{I} \\ 1 \leq \iota \leq m}} \{energy(i, \tau) + E(i+1, \Delta^1, \ldots, \Delta^\iota \setminus \tau, \ldots, \Delta^m) \mid \tau \subseteq \Delta^\iota \wedge (\tau_{start} = \Delta^\iota_{start} \vee \tau_{end} = \Delta^\iota_{end}) \}.$$

Once again, we skip the details of the notation used, since it was first introduced in section 3.1.1. The base cases are, also, easily determined from the above relation. Solving this dynamic program takes $O(mn^{4m+5})$ time, since, in the worst case, we have to compute $O(n^{4m+1})$ entries and, for each entry, the algorithm has to minimise over $O(mn^4)$ different possibilities. If m is a constant, then this shows that this particular problem is still in P.

Note. Unlike the original dynamic programming solution, we don't need to worry about partitioning the set of remaining jobs, when computing each entry of the dynamic programming table: there is only one recursive call and the time windows of each processor are never split up into additional ones. Nevertheless, it is easy to see that the overlapping sub-structure property[†] is still present.

[†]See section 3.1.1 for a more detailed explanation of this concept.

Chapter 4

A different scheduling problem

Many problems in Computer Science have a very similar structure, despite of having different formulations. Because of this property, we can use a very useful tool, by the name *reduction*, which allows us to transform an unknown problem into one we understand better – we might know its solution, complexity, etc. This is the basis for today's study and analysis of the field Computational Complexity Theory[1]. Because some problems can be reduced to others, we can categorise them, based on their relative difficulty (or, complexity). Finding reductions to different problems is an interesting way of tackling a problem difficult to solve and for whom we have no idea of its difficulty. Often, a good reduction scheme manages to solve an open problem (or prove it is unsolvable), but it also allows us to find solutions with better time/space complexities or stronger approximation ratios.

Unfortunately, when two problems don't share enough properties, finding reductions can be very hard or not possible at all. Nevertheless, we can always take advantage of any similarity between two problems. Building on this, Chien-Chung Huang and Sebastian Ott [22] found a connection between the non-preemptive speed scaling problem (or a special instance of it) to a variant of the well known *multi-processor scheduling problem* (MSP) analysed by Garey and Johnson [17]. The connection results from adopting some of the ideas used in the solution to the second problem, to design a QPTAS[2] that can (approximately) solve laminar instances of the non-preemptive speed scaling problem.

The solution from which Huang and Ott based themselves is a PTAS, so it would be interesting to see if it is possible to design an exact and faster algorithm for a more restricted version of this problem – this strategy could even give rise to faster approximation algorithms. This chapter explores this idea and it presents an optimal algorithm, along with a detailed proof of correctness. Unfortunately, there was no time to further extend

[1] One of the well known textbook in this field is the one written by Garey and Johnson, "Computers and Intractability: A Guide to the Theory of NP-Completeness" [17].

[2] Quasi-polynomial-time approximation scheme. Slower than a PTAS: its running time is $n^{\text{polylog}(n)}$ for each fixed $\epsilon > 0$.

the results of this chapter to more general instances, or to apply them in the original problem of this thesis. Nevertheless, this seems to be a promising strategy and something that is definitely worth looking at in the future.

4.1 The problem

The problem studied here is a generalisation of MSP [17], whose formulation is: given a set J of jobs and a set M of processors, the goal is to find a schedule of all jobs in J that minimises the maximum job completion time C_{max}. This is equivalent to minimising the maximum makespan for all processors in M. Each processor executes (without interruptions) at most one job j at a time, whose *size* – processing time – is p_j units. Unlike the speed scaling problem, there is no restriction on when a job can be executed and we can no longer control the speed of the processor (assume that it's constant). It is well known that this problem (MSP) belongs to the NP-hard complexity class.

It might seem counterintuitive, but we are, actually, interested in a harder version of the problem: one in which jobs might not be allowed to be scheduled in every processor. Following the notation used by Muratore, Schwarz and Woeginger in [29], this problem is denoted by *scheduling with job assignment restrictions*, or $R \mid p_{ij} \in \{p_j, \infty\} \mid {C_{max}}^{(1)}$, using the 3-field classification introduced by Graham, Lawler, Lenstra and Kan in [21]. This particular problem generalises MSP in the sense that each job $j \in J$ can only be scheduled in one of the machines[(2)] given by $\mathcal{M}(j) \subseteq M$. As such, we know that this problem is at least NP-hard, but we lack complete knowledge of its approximability. So far, the strongest approximation algorithm was designed by Lenstra, Shmoys and Tardos [27], whose approximation ratio is 2, even for the more general problem $R \mid\mid C_{max}$. In the same paper, the authors show that, for $R \mid p_{ij} \in \{p_j, \infty\} \mid C_{max}$, it is not possible to design a polynomial-time algorithm whose approximation ratio is better than 3/2 (unless P=NP). Unfortunately, this result still holds for the special case where, for all jobs $j \in J$, we have that $|\mathcal{M}(j)| = 2$ (see [16]). It remains an open problem do determine what, exactly, is the best approximation ratio we can get.

Special instances and goal overview. Despite the negative results mentioned above, there is a PTAS[(3)] for an interesting special instance of the problem, in which we have *nested machine sets* (see [29]). In this special case (first considered by Glass and Kellerer

[(1)] $p_{ij} \in \{p_j, \infty\}$ handles the fact that jobs can only be scheduled in specific machine sets

[(2)] For consistency, we will use the same notation introduced in [29] and use the term "machine" in place of "processor"

[(3)] See section 1.3 for an explanation of this term.

[18]), we have that, for any two jobs $i, j \in J$:

$$\text{either } \mathcal{M}(i) \subseteq \mathcal{M}(j) \ , \ \mathcal{M}(i) \supseteq \mathcal{M}(j) \text{ or } \mathcal{M}(i) \cap \mathcal{M}(j) = \varnothing.$$

As mentioned in [29], an application of this special recursive structure of the machine sets appears in the "drying stage of flour mills in the United Kingdom" (see [19]). This property of the machine sets is similar to the one in laminar instances (see section 1.4.2.1) of the non-preemptive speed scaling problem $(S \mid r_j, d_j \mid E)$. As indicated in the beginning of this chapter, Huang and Ott explored this similarity, to design a QPTAS for laminar instances in the non-preemptive speed scaling problem [22].

With the goal of further exploiting the resemblance of both problems, I explored the idea of designing an exact algorithm for $R \mid p_{ij} \in \{p_j, \infty\} \mid C_{max}$ using nested machine sets, which improves the running time for instances that are restricted by the number of different job sizes. This chapter presents an exact algorithm for the case that $p_j \in \{1, 2\}$ (for all jobs $j \in J$) and it gives an elaborate proof of correctness. For clarity and ease of understand, we consider first a special case of nested machine sets, in which, for any two jobs $i, j \in J$:

$$\text{either } \mathcal{M}(i) \subseteq \mathcal{M}(j) \text{ or } \mathcal{M}(i) \supseteq \mathcal{M}(j).$$

In this special case (first considered by Hwang, Chang and Lee [23]), we say that machine sets $\mathcal{M}(j)$ are *totally ordered* by inclusion. As in the case that machine sets are nested, it is easy to see a resemblance to purely laminar instances on the non-preemptive speed scaling problem. This special structure of the machine sets is, by itself, of scientific interest: as stated in [29], one of its possible applications is the assignment of computer programs (which require some amount of memory) to processors that have a limited memory capacity. Naturally, a job j can be assigned to any processor that can schedule a "lighter" job i, and thus, $\mathcal{M}(i) \subseteq \mathcal{M}(j)$. Generalisation to nested machine sets is accomplished in section 4.3.

4.1.1 Further notation

In order to simplify the following definitions, consider that all sets $\mathcal{M}(j)$ for all jobs j in J are elements of a family F of machines sets such that:

$$F_1 \subseteq F_2 \subseteq \ldots \subseteq F_{|F|}.$$

Definition 5 (Ranking of jobs)**.** The *rank* of a job j increases with the size of $\mathcal{M}(j)$:

$$\begin{aligned} &\text{rank} : J \to \mathbb{N} \\ &\text{rank}(j) \mapsto k, \text{ if } \mathcal{M}(j) = F_k. \end{aligned}$$

Similarly, we define below the ranking of a machine in M. For simplicity, we will use the same function operator *rank*, used for jobs.

Definition 6 (Ranking of machines)**.** The *rank* of a machine m is formally defined as:

$$\begin{aligned} &\text{rank} : M \to \mathbb{N} \\ &\text{rank}(m) \mapsto k, \text{ if } \mathcal{M}'(m) = F_k, \end{aligned}$$

where $\mathcal{M}'(m) = \underset{f \in F}{\text{argmin}} \{|f| : m \in f\}$.

A schedule S is a function $S : M \to \mathcal{P}(J)$, where $\mathcal{P}(J)$ is the power set of J. S_m denotes the set of jobs scheduled in machine m.

The makespan of a machine m under a schedule S is given by:

$$\gamma_S(m) = \sum_{j \in S_m} p_j.$$

If we can derive the schedule S from the context we simply denote the makespan by $\gamma(m)$.

The quality of a schedule is given by its maximum makespan:

$$\Gamma(S) = \max_{m \in M} \{\gamma_S(m)\}.$$

Finally, let jobs(S) denote the set of jobs scheduled in S.

Given the amount of symbols and operators, a summary of the overall notation is given in table 4.1.

Symbol / operator	Meaning
M	set of machines
J	set of jobs
F	set of families of machines
$\mathcal{M}(j) \in F$	machines where j can be scheduled
$\mathcal{M}'(m) \in F$	smallest set element of F that contains m
p_j	size (processing time) of job j
$\text{rank}(j)$	rank of job j
$\text{rank}(m)$	rank of machine m
S_m	set of jobs scheduled in machine m under a schedule S
$\gamma_S(m)$	makespan of machine m under a schedule S
$\Gamma(S)$	maximum makespan in schedule S
$\text{jobs}(S)$	set of jobs scheduled in S

Table 4.1: Summary of the notation introduced in section 4.1.

4.2 The solution

The algorithm described in this section follows an almost greedy strategy: it schedules jobs in a specific order, but it might need to re-assign previously scheduled jobs to different machines. It starts by initializing a schedule where no job is scheduled. The scheduling order of the jobs is given by the non-decreasing order of their ranks (breaking ties arbitrarily) and we schedule one job k at a time. If k has size 1 or if we can schedule k without increasing the maximum makespan, then we schedule it on any machine $m \in \mathcal{M}(k)$ of minimum makespan. Otherwise, we construct a schedule for each possible assignment of k, by re-scheduling previously assigned jobs. From this set of schedules, we keep the best one (the one that has less maximum makespan), under which we schedule the next job and repeat the whole process until all jobs were scheduled.

A summary of the whole algorithm is given below. The schedule being constructed is represented by S and the term "unschedule" is used to denote the removal of a job from the set of scheduled jobs of some machine.

Algorithm $\mathcal{A}$

1. Let S be an empty schedule

2. WHILE there are unscheduled jobs, DO

(a) Pick any unscheduled job k of minimum rank

(b) Pick any machine $m \in \mathcal{M}(k)$ of minimum rank

(c) IF $p_k = 1$ or $\gamma(m) + p_k \leq \Gamma(S)$, THEN

i. Schedule, under S, job k in machine m

(d) ELSE

i. $S^{new} \leftarrow \arg\min_{i \in \mathcal{M}(k)} \{ \Gamma(\textbf{Re-schedule}(S, k, i)) \}$

ii. $S \leftarrow S^{new}$

3. RETURN S

The obvious reason for considering jobs in non-decreasing order of their ranks is the benefit that comes from, successively, handling larger machine sets $\mathcal{M}(k)$: we consider the more "restricted" jobs first, i.e., the jobs that provide less freedom in the scheduling choice.

We now give the details on how to build the schedules in step 2.(d) of alg. $\mathcal{A}$. As we can see, this is done according to a function **Re-schedule** that receives the following parameters:

- the current schedule S
- a machine i
- a job k

This routine is called several times, once for each possible value of $i \in \mathcal{M}(k)$. It begins by scheduling job k on machine i and removing as many jobs of size 1 as necessary in order to prevent the increase of i's makespan. We denote the set of these removed jobs by T. Details of its computation are given below.

Procedure Compute T

1. Let $T \leftarrow \{\}$

2. Let γ' be i's makespan before scheduling job k on it

3. WHILE $\gamma(i) > \gamma'$ and i has jobs of size 1, DO

- **(a)** Let j be any job of size 1 scheduled in i
- **(b)** Unschedule j from machine i
- **(c)** $T \leftarrow T \cup \{j\}$

After, the function determines the set T^* of the jobs of maximum possible rank that can be obtained by successively exchanging one job from T with another one from the set of scheduled jobs (in some machine). This exchange implies one job from T to be scheduled in the machine where we took the other exchanged job. Note that such exchange is only possible if the job coming from T is allowed to be scheduled on the desired machine. Naturally, we only trade jobs if the minimum rank in T increases.

There are many ways to compute T^*, one of them is given below.

Procedure Compute T^*

(**Compute** T)

1. Let $T^* \leftarrow \{\}$

2. Let $M' \leftarrow \mathcal{M}(k)$

3. WHILE $M' \neq \varnothing$ and $T \neq \varnothing$, DO

- **(a)** Remove any machine m' of minimum rank from M'
- **(b)** Move every job j from T to T^*, if $m' \notin \mathcal{M}(j)$
- **(c)** **Update** T

4. $T^* \leftarrow T^* \cup T$

Update T is the routine responsible for the job exchange mentioned above, in the sense that T gets the maximum ranked jobs of the set $T \cup \{\text{jobs of size 1 scheduled in } m'\}$. Moreover, since we are exchanging jobs, we make sure that T doesn't change its size. Following is a possible implementation of this step:

Procedure Update T

1. Let $T_{new} \leftarrow \{\}$ and $R \leftarrow T \cup \{\text{jobs of size 1 scheduled in } m'\}$
2. REPEAT $|T|$ times:
   - **(a)** Let r be any job of maximum rank of R
   - **(b)** Move r from R to T_{new}
3. Schedule all jobs $T \setminus T_{new}$ in m'
4. Unschedule all jobs $T_{new} \setminus T$ from m'
5. $T \leftarrow T_{new}$

Steps 3. and 4. ensure us that we do all the re-schedules required to update T.

Finally, the routine **Re-schedule** (re)schedules every job in T^* by following the same strategy used in the main algorithm: sequentially assigning jobs of non-decreasing rank to machines of minimum makespan. Since all jobs in T^* have size 1, we know that we won't need to call **Re-schedule** for this task.

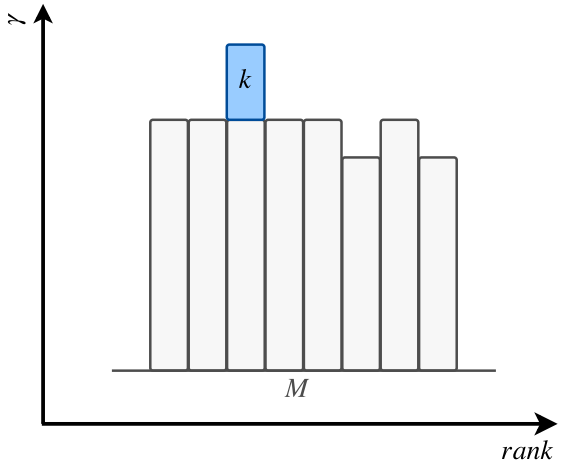
An overview of the whole routine is presented below and illustrated in fig. 4.1.

Function Re-schedule

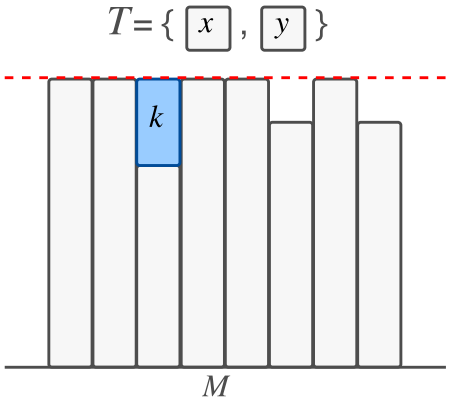
Input: schedule S, job k, machine i
Output: schedule S (modified)

1. Update S by scheduling job k on machine i
2. **Compute** T
3. **Compute** T^*
4. Update S by scheduling all jobs of T^* using the strategy of the main algorithm
5. RETURN S

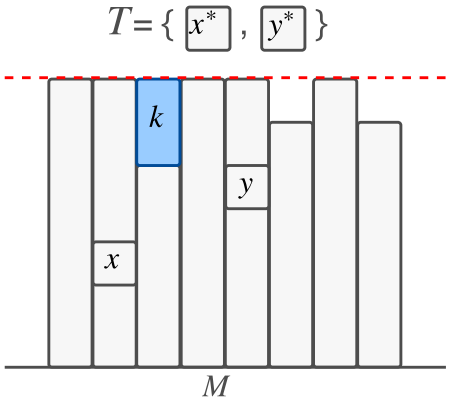
Observe that steps 2. and 3. make implicit changes in schedule S. See above their implementations for details.



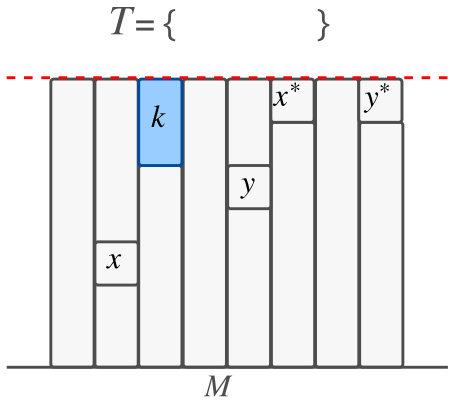
(a) Initial schedule for some of the machines in M, conveniently ordered by increasing rank.



(b) Step 2.



(c) Step 3.



(d) Step 4.

Figure 4.1: Illustration of the routine **Re-schedule** for job k.

Note. The reader might think: since we only re-schedule jobs of size 1, why not first schedule all jobs of size 2 in the same order and, only then, schedule jobs of size 1? A simple counter-example of the optimality of this strategy is illustrated in fig. 4.2. Moreover, the greedy strategy of successively scheduling every job of non-decreasing rank in machines of minimum makespan does not work as well, see fig. 4.3. The routine **Re-schedule** is needed in this case.

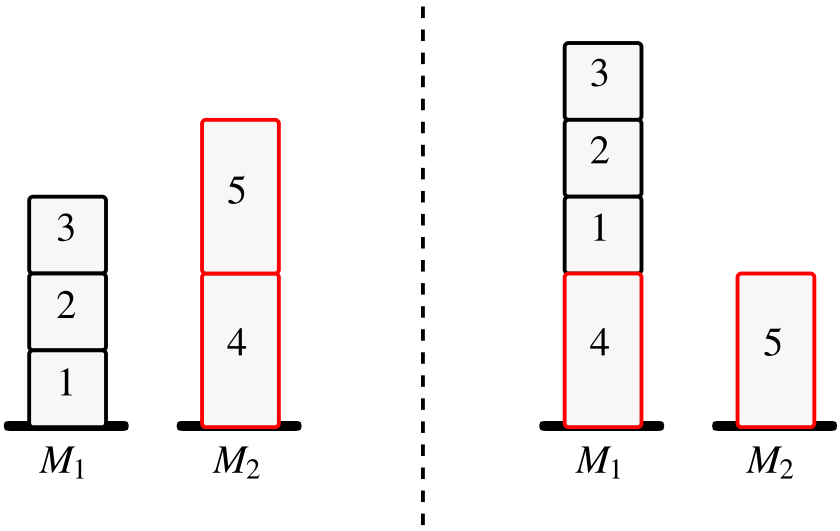


Figure 4.2: Counter-example for the optimality of the strategy of scheduling all jobs of size 2 before the jobs of size 1 – right schedule. On the left we can see the optimal solution. We assume that $\mathcal{M}(1) = \mathcal{M}(2) = \mathcal{M}(3) = \{M_1\}$ and that $\mathcal{M}(4) = \mathcal{M}(5) = \{M_1, M_2\}$. Red jobs have size 2 and black jobs have size 1.

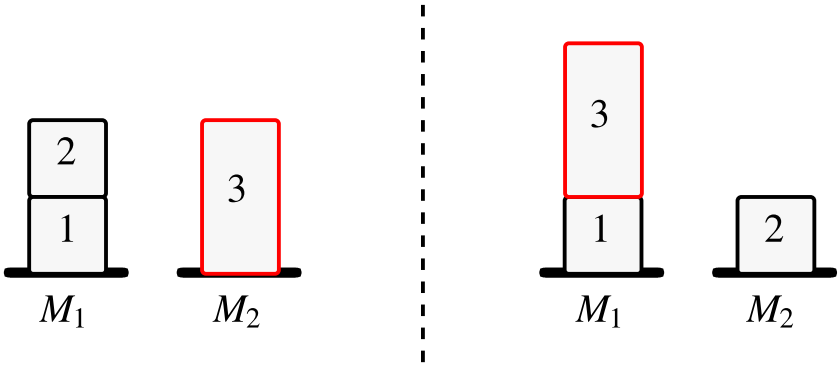


Figure 4.3: Counter-example for the optimality of the strategy of scheduling all jobs by increasing order of their ranks (no matter their sizes) – right schedule. On the left we can see the optimal solution. We assume that $\mathcal{M}(1) = \mathcal{M}(2) = \mathcal{M}(3) = \{M_1, M_2\}$ and that jobs are considered in the order given by their identification number. Red jobs have size 2 and black jobs have size 1.

4.2.1 Correctness of algorithm $\mathcal{A}$

We begin by confirming the termination of the algorithm.

Claim 1. *Algorithm $\mathcal{A}$ terminates after $|J|$ iterations of its main loop.*

Proof. In each iteration of the main loop in alg. $\mathcal{A}$, exactly one new job is scheduled. Since we have $|J|$ jobs in total, the algorithm stops after $|J|$ iterations. □

Now, we demonstrate the correctness of $\mathcal{A}$ with a proof by induction, assuming, for now, that its inductive step holds.

Theorem 3. *Algorithm $\mathcal{A}$ builds an optimal schedule to problem $R \mid p_{ij} \in \{p_j, \infty\},\ p_j \in \{1, 2\} \mid C_{max}$ using ordered machine sets.*

Proof. We show that the algorithm is optimal by proving by induction on i that the $i+1^{\text{th}}$ iteration of the algorithm produces an optimal schedule. For the correctness of the inductive step it is enough to show that both steps 2.(c) and 2.(d) of the algorithm are correct, i.e., that both build an optimal schedule for all jobs picked up so far. Since this task is rather complex, it is split up into two parts that are postponed to sections 4.2.1.1 and 4.2.1.2, respectively. Thus, we assume, for now, that the inductive step is correct.

Claim 1 ensures us that the algorithm will eventually stop, so it remains to show that the base case for the proof is correct as well, i.e., when $i = 0$. This corresponds to the moment before the first job is scheduled, in which the schedule is empty. In other words, its maximum makespan is zero, which is, necessarily, optimal. □

4.2.1.1 Correctness of $\mathcal{A}$'s step 2.(c)

We begin by showing that the following lemma holds.

Lemma 3. *Let S' be an optimal schedule of jobs J' and S a schedule of jobs $J' \cup \{j\}$. If $\Gamma(S') = \Gamma(S)$, then S is also optimal.*

Proof. We prove by contradiction. If S is not optimal, then there is a schedule S^* with has even less maximum makespan. But, if we remove job j from S^*, we end up with a schedule for the same set of jobs J' that is better than S', a contradiction. □

We proceed by introducing another lemma that will be useful afterwards.

Lemma 4. *Let S' be a schedule of jobs J' over a set of machines M'. If, for every two machines i, j in M', we have that $|\gamma(i) - \gamma(j)| \leq 1$, then S' is an optimal schedule of jobs J'. The converse is not true.*

Proof. It is easy to see that, if preemption is not allowed, the maximum makespan of any schedule is always lower bounded by:

$$\Gamma(\hat{S}) \geq \text{LB}(\hat{J}, \hat{M}) := \left\lceil \frac{\sum_{j \in \hat{J}} p_j}{|\hat{M}|} \right\rceil,$$

for any schedule $\hat{S}$ of jobs $\hat{J}$ in machines $\hat{M}$.

Clearly, if the premise holds, then $\Gamma(S') = \text{LB}(J', M')$, which makes S' an optimal schedule of jobs J' over the set of machines M'. □

Finally, we are able to prove the correctness of step 2.(c). As in algorithm $\mathcal{A}$, we denote by k the job that we are trying to schedule on each iteration.

Theorem 4. *Step 2.(c) of algorithm $\mathcal{A}$ builds an optimal schedule S for job k and all the previously scheduled jobs.*

Proof. Let us assume that, in the end of this step, the maximum makespan increased. If it didn't, then, by lemma 3, S is optimal. Observe that step 2.(c) is only executed when (i) job k has size 1 or (ii) when it is possible to directly schedule k without increasing the maximum makespan. Hence, it must be the case that only the former condition (i) holds, given the initial assumption. What's more, all machines in $\mathcal{M}(k)$ have the same (maximum) makespan. Thus, whatever machine is chosen by the algorithm to schedule k, will be an optimal choice. This follows directly from lemma 4. □

4.2.1.2 Correctness of $\mathcal{A}$'s step 2.(d)

In this case, the algorithm finds an optimal schedule for job k (the one we are trying to schedule on the current iteration of the algorithm), by calling the routine **Re-schedule** for every machine allowed to schedule k. In order to prove the correctness of this step, we need to make a substantial assumption: there is a solution for scheduling k and all previously assigned jobs that does not require re-scheduling any job of size 2. This is formalised in the theorem below (recall from section 4.1.1 that S_x denotes the set of jobs scheduled in machine $x \in M$, under schedule S).

Theorem 5. *Given an optimal schedule S' of jobs J', there always exists an optimal schedule S of the jobs $J' \cup \{k\}$ (with rank$(k) \geq \max_{j \in J'}\{\text{rank}(j)\}$) that does not re-schedule jobs of size 2 in S'. That is, for all jobs $j \in J'$:*

$$j \in S'_y \implies j \in S_y, \quad \textit{if } p_j = 2$$

Its proof is delayed to section 4.2.3, given its complexity. For now, let us assume that the theorem holds. For simplicity, we will adopt the same notation introduced in the theorem's statement, i.e., S represents the optimal schedule after assigning job k (which is the job picked by the alg. in the current iteration) and S' is the schedule built by the algorithm in the previous iteration. The jobs scheduled in S' and S are, respectively, given by J' and $J' \cup \{k\}$. Consider, also, that k was scheduled in machine m under the schedule S.

We want to show, by contradiction, that S is optimal. To that extent, assume, for the rest of the proof, that S isn't optimal, but some schedule S^* is. Given theorem 5, we can presume that S^* has the same schedule for jobs of size 2 as the one in S'. Moreover, consider that $\Gamma(S') < \Gamma(S)$, or otherwise, we can finish this proof by applying lemma 3 (under section 4.2.1.1).

Let us partition the set of machines $\mathcal{M}(k)$ into three subsets M_1, M_2 and M_3 according to the requirements specified in table 4.2. For future reference, whenever an expression similar to "job is scheduled in M_i" is used, it naturally means (in a handier way) that the job is scheduled in some machine of M_i.

M_1	$\Gamma(S) - \gamma_S(x) = 0$, for all x in M_1
M_2	$\Gamma(S) - \gamma_S(x) = 1$, for all x in M_2
M_3	$\Gamma(S) - \gamma_S(x) > 1$, for all x in M_3

Table 4.2: Partition of the machines set $\mathcal{M}(k)$

Clearly, $|M_1| > 0$, since there must be at least one machine whose makespan is maximum. In addition, let us consider that $|M_3| > 0$, or otherwise S would be an optimal schedule for jobs $J' \cup \{k\}$ over the set of machines $\mathcal{M}(k)$ (lemma 4).

Intuitively, we can think of S^* as a more balanced version of the schedule S, i.e., with less discrepancy on the makespans of the machines of $\mathcal{M}(k)$. In fact, it is easy to see that the amount of "work" scheduled in all machines of M_3 should be bigger under schedule S^*, compared to S (see fig. 4.4).

In this context, we are going to find contradictions while attempting to transform S into S^*. This transformation is, certainly, possible if we simply re-schedule jobs on S. Besides, we know that at least one job scheduled (under S) in M_1 or M_2 must be transferred over to M_3, otherwise S is optimal. Figure 4.4 illustrates this point. In fact, the only way of decreasing the maximum makespan of S is by removing at least one job from a machine of maximum makespan and making successive job exchanges until we get a job that is sufficiently ranked to be transferred to M_3. A formal explanation of this idea follows (definition 7).

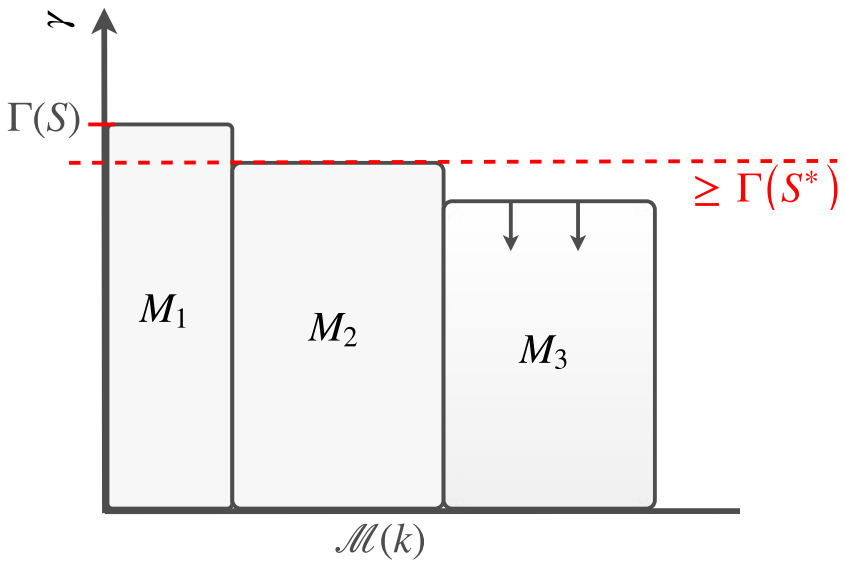


Figure 4.4: Partition of the set of machines $\mathcal{M}(k)$ (into M_1, M_2 and M_3) according to their makespan in a schedule S. The red dashed line indicates the maximum possible makespan of an optimal schedule S^*, assuming that schedule S isn't optimal. Clearly, some jobs scheduled (under S) in M_1 or M_2 must be re-scheduled to M_3, in order to reduce the maximum makespan.

Definition 7 (Re-schedule sequence)**.** A re-schedule sequence (π_M, π_J, l) is a series of l feasible re-schedules defined by a permutation $\pi_M : \{1, \ldots, l+1\} \to M$ of machines and a permutation $\pi_J : \{1, \ldots, l\} \to J$ of jobs such that job $\pi_J(i)$ is re-scheduled from machine $\pi_M(i)$ to machine $\pi_M(i+1)$, for all $1 \leq i \leq l$, and as long as $\pi_M(i+1) \in \mathcal{M}(\pi_J(i))$.

The idea mentioned above becomes easier to grasp now. An illustration of a re-schedule sequence is given in fig. 4.5.

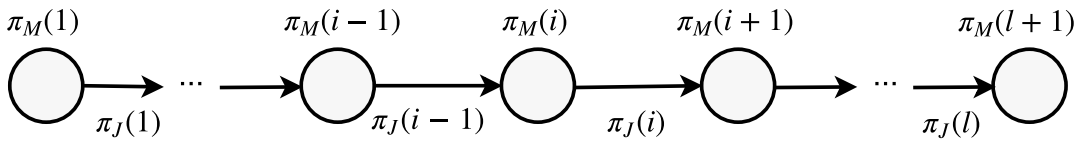


Figure 4.5: Illustration of a re-schedule sequence (π_M, π_J, l) (see definition 7). Nodes correspond to machines $\pi_M(i)$ for all $1 \leq i \leq l+1$ and arrows correspond to the re-schedule of a job $\pi_J(i)$ for all $1 \leq i \leq l$.

Lemma 5. *If S is not optimal we need at least one re-schedule sequence (π_M, π_J, l) where $\pi_M(1) \in M_1$ and $\pi_M(l+1) \in M_3$ in order to be able to transform S into S^*.*

Proof. It is easy to see that, in order to decrease the maximum makespan of S one needs to remove jobs from M_1, so consider all re-schedule sequences that start on a machine of M_1. If all of them end in a machine outside of M_3, then the last re-scheduled job of those sequences ends up in either M_1 or M_2, therefore not decreasing the maximum makespan. Hence, there must be at least one re-schedule sequence (π_M, π_J, l), such that $\pi_M(1) \in M_1$ and $\pi_M(l+1) \in M_3$. □

We now proceed by making another observation, regarding the jobs scheduled in machines of M_1, under S. Recall that T^* is the set of jobs (computed in step 3. of the function **Re-schedule**) of maximum possible rank that can be obtained by successively exchanging jobs between $T^{(1)}$ and some machine.

Lemma 6. *Every machine in M_1 schedules (under S) at least one job from the set T^* if $T^* \neq \varnothing$.*

Proof. Recall our assumption (from the beginning of this section) that $\Gamma(S') < \Gamma(S)$, where S' is the schedule built by the same algorithm in the previous iteration. This implies that all machines in M_1 increased their makespan in schedule S. According to the algorithm, a machine only increases its makespan by either:

(i) scheduling a job from T^*;
(ii) scheduling job k (of size 2) and not removing enough jobs of size 1 from the same machine (i.e., when $|T| < 2$);
(iii) or by both (i) and (ii).

The second case (ii) is the only one that, potentially, contradicts the lemma statement. However, we know that this case doesn't hold for the machines of M_1. If it did, we could never decrease the makespan of these machines without re-scheduling jobs of size 2. But we know that there is a schedule (e.g. S^*) where the same machines have a smaller makespan and schedule the same jobs of size 2 (see theorem 5). Hence, by contradiction, case (ii) doesn't hold for the machines of M_1 and, consequently, none of the possibilities mentioned contradicts the lemma statement. □

Having proved lemma 6, we move on to check that the set T^* is correctly computed under algorithm $\mathcal{A}$.

Lemma 7. *For every job $t \in T^*$, t has the highest rank over all jobs scheduled in the set of machines $\mathcal{M}(t)$, under S.*

[1] T is initially the set of jobs taken out from the machine where job k was scheduled.

Proof. By contradiction, if there was a job t' such that $\text{rank}(t') > \text{rank}(t)$ (for some job t in T^*) and t' is scheduled in some machine x of $\mathcal{M}(t)$, then either:

(i) the algorithm failed to select the jobs of maximum rank in step 2.(a) of the routine **Update** T;

(ii) or: $x \notin \mathcal{M}(k)$, the set of machines covered by the procedure **Compute** T^*.

Naturally, we assume that the algorithm runs as expected, so the first case never happens. In the latter, we also have a contradiction, because its statement implies that $\mathcal{M}(t) \not\subset \mathcal{M}(k)$, which we know it can't be true: given the scheduling order of the jobs, we have that $\text{rank}(t) \leq \text{rank}(k)$ and, by the definition of rank, this indicates that $\mathcal{M}(t) \subseteq \mathcal{M}(k)$. □

Finally, we are in conditions of proving the correctness of step 2.(c).

Theorem 6. *Step 2.(d) of algorithm $\mathcal{A}$ builds an optimal schedule for job k and all the previously scheduled jobs.*

Proof. Recall from the beginning of this section that we are assuming that the solution built from alg. $\mathcal{A}$, denoted by S, is not optimal and that we partition $\mathcal{M}(k)$ into sets M_1, M_2 and M_3. Also, we are presuming that $\Gamma(S') < \Gamma(S)$.

Let $R = (\pi_M, \pi_J, l)$ be a re-schedule sequence equivalent to the one mentioned in lemma 5, where: $\pi_M(1) \in M_1$ and $\pi_M(l+1) \in M_3$.

The idea for proving that this theorem holds is to show that the very existence of R leads to contradictions on the behaviour of algorithm $\mathcal{A}$. To this extent, we distinguish between two complementary cases. Let j^* denote the highest ranked job scheduled in $\pi_M(1)$. Then, either j^* can be scheduled in $\pi_M(l+1)$ (case 1) or not (case 2). By combining lemmas 6 and 7, we know that $j^* \in T^*$, if $T^* \neq \varnothing$. We ignore the case of T^* being empty, because when that happens, it is easy to see that S must be optimal. If $T^* = \varnothing$, then machine m (where we scheduled k), doesn't have any jobs of size 1 and, therefore, its makespan is the same in both S and S^* (recall our assumption of S^* and theorem 5, in the beginning of this section). Thus, given that, besides scheduling k, no other change was made to the optimal schedule S' (the one built in the previous iteration of the algorithm), it must be the case that S is optimal.

Below we can see the analysis made to each of the cases mentioned above, considering that $T^* \neq \varnothing$.

Case 1. $\pi_M(l+1) \in \mathcal{M}(j^*)$ $\quad$ $(T^* \neq \varnothing)$

Clearly, j^* was the last job assigned to $\pi_M(1)$. This follows from the fact that the jobs in T^* are scheduled in non-decreasing order of their rank (step 4. of the routine **Re-schedule**). But we know that, before the insertion of j^*, we had that:

$$\gamma(\pi_M(1)) - \gamma(\pi_M(l+1)) \geq 1,$$

since $\pi_M(1) \in M_1$ and $\pi_M(l+1) \in M_3$. Thus, we have a contradiction, because $\pi_M(1)$ was the machine of minimum makespan incorrectly chosen by the algorithm to schedule j^*. Therefore, it can never happen that $\pi_M(l+1) \in \mathcal{M}(j^*)$.

Case 2. $\pi_M(l+1) \notin \mathcal{M}(j^*)$ $\quad$ $(T^* \neq \varnothing)$

In this case, it is clear that, at some point in the successive re-schedules of R, we traded a job by another one that has a rank greater than $\text{rank}(j^*)$. In other words, for some $i \in \{1, \ldots, l\}$, we have that:

(i) $\text{rank}(\pi_J(i)) > \text{rank}(j^*)$ and
(ii) $\text{rank}(\pi_J(i')) \leq \text{rank}(j^*)$, for all $1 \leq i' < i$.

If there was no such i, then R, as a feasible re-schedule sequence, wouldn't be able to send a job over to M_3. See fig. 4.6 for an illustration.

Given (ii) and the fact that $\pi_M(i) \in \mathcal{M}(\pi_J(i-1))$ (by definition of re-schedule sequence – definition 7), we know that $\pi_M(i) \in \mathcal{M}(j^*)$. Thus, there is a job (e.g. $\pi_J(i)$), scheduled (under S) in a machine of $\mathcal{M}(j^*)$ with a rank greater than $\text{rank}(j^*)$. This, clearly, contradicts lemma 7 (recall that $j^* \in T^*$). Hence, we cannot afford to have that $\pi_M(l+1) \notin \mathcal{M}(j^*)$.

Therefore, we conclude that there can't be a re-schedule sequence that transforms S into the optimal schedule S^* in which $\pi_M(1) \in M_1$ and $\pi_M(l+1) \in M_3$. As we have seen with both cases above, that could only happen if S wasn't built from algorithm $\mathcal{A}$. On the assumption that S isn't optimal, this clearly contradicts lemma 5. Thus, we can infer that S is indeed optimal. □

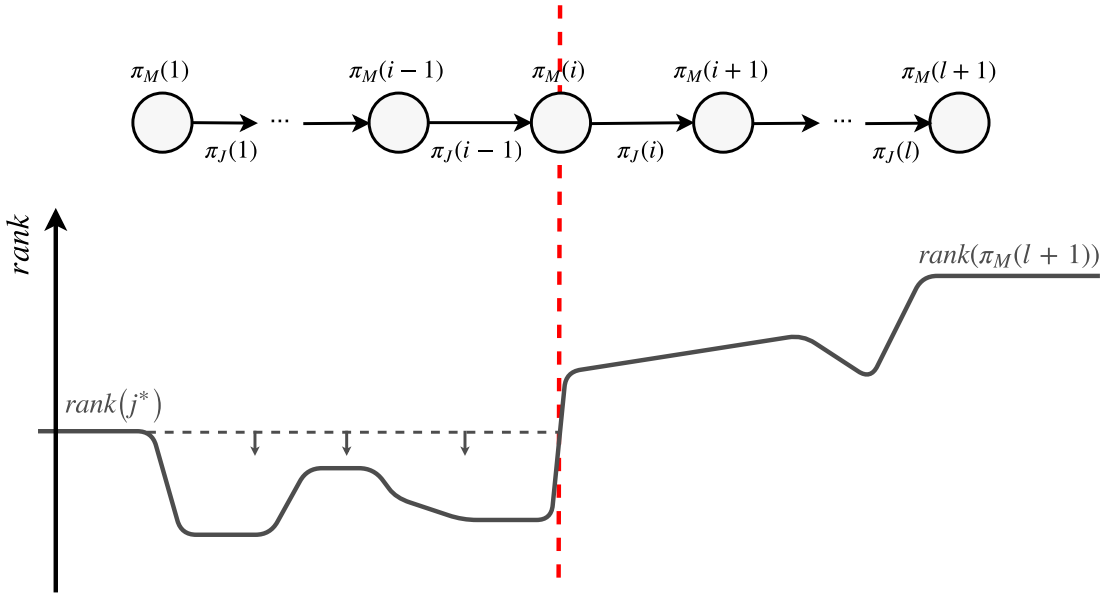


Figure 4.6: Illustration of case 2 from proof of theorem 6. The sequence of connected nodes on the top denotes a re-schedule sequence (see definition 7) $R = (\pi_M, \pi_J, l)$ and j^* is the highest ranked job scheduled in $\pi_M(1)$. If $\pi_M(l+1) \notin \mathcal{M}(j^*)$, then, at some point in R (red line), we must trade off some job by one whose rank is bigger than rank(j^*) (in this case, $\pi_J(i)$).

4.2.2 Complexity

Time complexity. We already know, by claim 1, that alg. $\mathcal{A}$ terminates in $|J|$ iterations. In each iteration of the algorithm, we spend, in the worst case, $O(m^2)$ time. This happens when we need to schedule a job of size 2 and call the routine **Re-schedule**. In this case, we try to schedule the new job k in every allowed machine (at most m times) and, for each attempt, we loop again over the set of machines under the procedure **Compute** T^*. Each iteration of this loop can be easily done in $O(1)$ time, if we use sorted data structures for the sets manipulated in the routine **Update** T[(1)]. The reason is that we only insert (or remove) elements of maximum value for each set, including the ones responsible for scheduling jobs. Moreover, only a constant number of operations is done, since $|T| = O(1)$. The remaining steps of the function **Re-schedule** can, clearly, be done in $O(m)$ time, which doesn't increase (asymptotically) the final time complexity.

However, it takes $O(n \log n)$ to sort the sets of jobs and machines by their rank (assuming $n \geq m$). Hence, the overall time complexity is:

$$O(n(\log n + m^2))$$

[(1)]The sets are: T, R, T_{new} and S_i for scheduling (removing) jobs in (from) machine i

Space complexity. The algorithm requires $O(n)$ of extra space[1]: $O(n)$ for sets S_m for all $m \in M$ and $O(1)$ for T and all auxiliary variables.

4.2.3 Proof of theorem 5

Theorem 5. *Given an optimal schedule S' of jobs J', there always exists an optimal schedule S of the jobs $J' \cup \{k\}$ (with rank$(k) \geq \max_{j \in J'}\{\text{rank}(j)\}$) that does not re-schedule jobs of size 2 in S'. That is, for all jobs $j \in J'$:*

$$j \in S'_y \implies j \in S_y, \quad \text{if } p_j = 2$$

Recall from section 4.1.1 that S_x denotes the set of jobs scheduled under machine $x \in M$.

Unfortunately, the proof of this theorem is quite complex. Its argument relies on a transformation between schedules, so we start by defining below a mechanism that accomplishes this.

Definition 8 (Re-schedule graph)**.** A *re-schedule graph* is a data structure that holds the necessary information to convert one schedule into another one that includes one additional job. More specifically, a *re-schedule graph* of a schedule S is a directed graph where each node corresponds to a machine in S and each arc $(a, b)_j$ to the "re-scheduling" of job j from machine a to b. Every arc in the graph as a colour: *red*, if the corresponding job has size 2, or *black*, otherwise. The scheduling of the new job k in a machine m is given by an additional node (that we will denote by node α) and an arc $(\alpha, m)_k$.

Formally, a *re-schedule graph* is a tuple $G = (V, A, k, m, S)$, where:

$$\boldsymbol{S} \text{ is a feasible schedule}$$
$$\boldsymbol{A} = \{(a, b)_j \mid j \in S_a \text{ and } b \in \mathcal{M}(j)\} \cup \{(\alpha, m)_k\}$$
$$\boldsymbol{V} = \{\text{all nodes with an incoming/outgoing arc in } A\} \cup \{\alpha\}$$
$$\boldsymbol{colour}((a, b)_j) = \begin{cases} \text{red}, & \text{if } p_j = 2 \\ \text{black}, & \text{otherwise.} \end{cases}$$

Let $\mathcal{A}(G)$ be the schedule after the *activation* of the re-schedule graph G. Activating a re-schedule graph corresponds to applying all re-schedules given by the arcs of the graph. Note that $\mathcal{A}(G)$ is always a feasible schedule by the above definition. If $\mathcal{A}(G)$ is optimal (regarding its scheduled jobs), then we say that G is an *optimal* re-schedule graph.

[1] We ignore the input sets J and M as well.

The idea of this proof is to transform any optimal re-schedule graph into one that doesn't re-schedule jobs of size 2. In other words, we want a graph whose only red arc is the one corresponding to job k. To make things easy, we start by transforming the original (unrestricted) graph into a simplified version of itself (see definition 11). Only then, we are in a good position to design a strategy for the final transformation.

Following are some basic definitions used throughout the section.

Definition 9 (Machine increment)**.** The *increment* of a machine is the increase (or decrease) in the makespan of that machine after activating a re-schedule graph for a given schedule. Formally:

$$I^G(m) = \gamma_{\mathcal{A}(G)}(m) - \gamma_S(m)$$

corresponds to the increment of a machine m according to a re-schedule graph G applied to a schedule S. If G can be derived from its context, we simply denote the increment by $I(m)$.

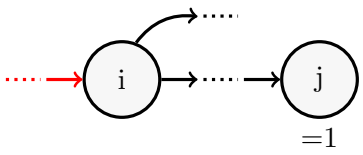
Definition 10 (Types of paths)**.** Let us denote a path in a graph by $i \rightarrow^* j$, where i and j are, respectively, the start and finishing nodes in that path. Additionally, we allow paths to be empty, that is, when $i = j$. A path denoted by $i \rightarrow^+ j$ is, necessarily, non-empty.

Let $P = i \rightarrow^* j$ be a path in a re-schedule graph G, in which: (i) all arcs are black, (ii) node i has at least one incoming red arc and (iii) $I(j) = 1$. We say that P is of *type*:

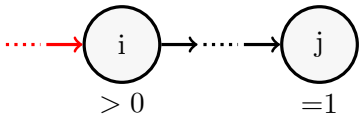
- **A**, if i has at least 2 outgoing black arcs.
- **B**, if P is not of type A and if $I(i) > 0$.
- **C**, if P is not of type A and if $I(i) < 0$.

An example illustrating each path type is given in fig. 4.7.

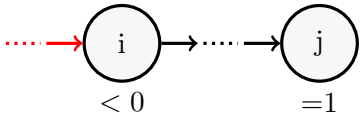
We are, now, able to introduce the kind of graph we would like to have after the first transformation mentioned above.



(a) Type A



(b) Type B



(c) Type C

Figure 4.7: Illustration of the three different types of paths in a re-schedule graph (see definition 10). Below each node, is its increment. For simplicity, the arcs that are not relevant were not drawn.

Definition 11 (Simplified graph)**.** A *simplified graph* G is a re-schedule graph with the following properties:

1. node α is the only source node in G;
2. G does not have *unicoloured cycles*, i.e., cycles whose arcs have all the same colour;
3. G does not have any path of type C.

The following claim will be useful for proving the upcoming lemma 8.

Claim 2. *Let G and G' be two re-schedule graphs of the same* optimal *schedule S. If, for every machine m in S, we have that either:*

$$\textbf{(i)} \quad I^G(m) \leq I^{G'}(m), \quad \textit{or}$$
$$\textbf{(ii)} \quad I^G(m) \leq 0,$$

then

$$\Gamma(\mathcal{A}(G)) \leq \Gamma(\mathcal{A}(G')).$$

Proof. Suppose $\Gamma(\mathcal{A}(G)) > \Gamma(\mathcal{A}(G'))$ and at at least one of the conditions hold. Let x be one machine in $\mathcal{A}(G)$ with the maximum makespan. We know that condition (i) does not hold for x and, thus, (ii) must hold. This means that:

$$\begin{array}{rccccccc} & \gamma_{\mathcal{A}(G')}(x) & < & \gamma_{\mathcal{A}(G)}(x) & \leq & \gamma_S(x) & & \\ \implies & \Gamma(\mathcal{A}(G')) & < & \Gamma(\mathcal{A}(G)) & \leq & \gamma_S(x) & \leq & \Gamma(S) \end{array}$$

This contradicts the optimality of S: since $\Gamma(\mathcal{A}(G')) < \Gamma(S)$, removing the added job in $\mathcal{A}(G')$ would result in a schedule better than S. Hence, it must be the case that $\Gamma(\mathcal{A}(G)) \leq \Gamma(\mathcal{A}(G'))$. □

Lemma 8. *Given an optimal schedule S' of jobs J', there always exists a simplified graph whose activation transforms it into an optimal schedule S of the jobs $J' \cup \{k\}$.*

Proof. It is easy to see that any re-schedule graph of a schedule S' can build S: for every job $i \in \text{jobs}(S')$, if $i \in S'_a$ and $i \in S_b$ (with $a \neq b$), add an arc $(a, b)_i$ to the (initially empty) graph. Finally, if $j \in S_m$, add an arc $(\alpha, m)_j$ and all the necessary nodes. Let G denote any such graph.

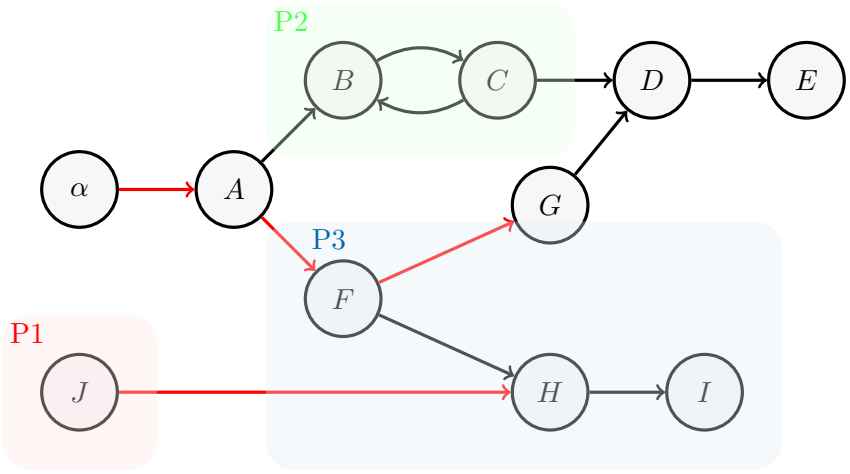
The proof will consist one gradually changing G to meet every property of a simplified graph, separately:

• Property 1. While there is a node $i \neq \alpha$ that is a source in G, remove it from the graph along with its outgoing arcs. Since i doesn't have any incoming arcs, its increment becomes null. When the only source node of G is m, we can stop.

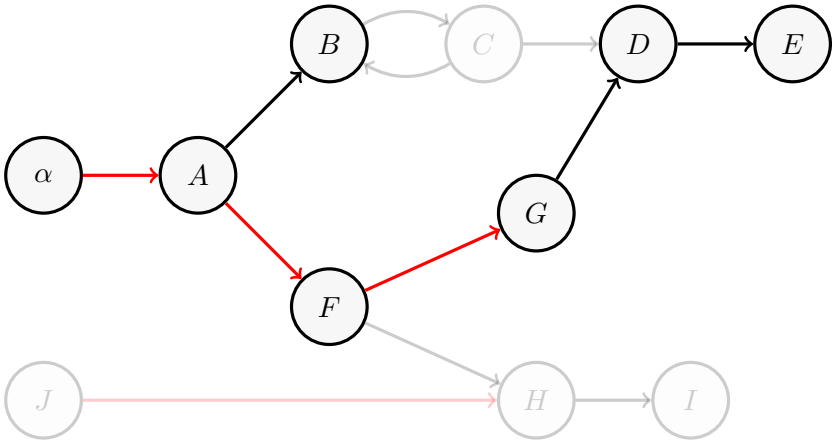
• Property 2. While there are unicoloured cycles in G, choose any one of them and remove all its arcs from G. If the graph is left with disconnected nodes, remove them as well. It is easy to see that, after each of these cycles removals, the increments of all machines in G don't change. To make sure property 1 is satisfied, repeat the previous step again.

• Property 3. While there exists a path $P = s \rightarrow^* t$ of type C, simply remove all arcs of P from G. If G is left with disconnected nodes, remove them as well. It is easy to see that, after removing the arcs, the only node in P that increases its increment is s. However, given the definition of a type C path, $I(s) < 0$ before the arcs removal. This means we can afford to remove at least 1 black outgoing arc of s and keep its increment non-positive.

After every transformation we are left with a simplified graph G that remains feasible, since we only removed arcs from a previous feasible graph. What's more, the increments that changed, remained non-positive in all transformations. Therefore, the schedule produced by G is still optimal by claim 2 (using G' as the graph before all transformations). An illustration of this transformation process is given in fig. 4.8. □



(a)



(b)

Figure 4.8: Example of a transformation of an unrestricted graph (a) into a *simplified* one (b). Notice how the graph on the top is violating the three properties inherent to simplified graphs (see definition 11).

Lemma 9. *Given a simplified graph $G = (V, A, k, m, S)$, all nodes with an increment of 1 are the finishing nodes of a path in G that has either type A or B.*

Proof. Let $j \in V$ be a node with $I(j) = 1$. It is easy to see that, if j doesn't have any black incoming arcs, it must have at least one black outgoing arc. In this case, j is the finishing node of an empty path of type B, which confirms the validity of the lemma. So, from now on, consider that j has at least one incoming black arc. Since there are no unicoloured cycles in G, there always exists a path P finishing on j that has at least one red arc. If no such path existed, then the source of G would only have black outgoing arcs. Clearly, this isn't the case because the definition of a simplified graph says that there exists a single source node α that only has one red outgoing arc. Hence, let i be the last node in P that has a red incoming arc. Since G doesn't have paths of type C, one of the following cases must hold:

(1) $I(i) > 0$;
(2) $I(i) = 0$ and i has a least two black outgoing arcs;
(3) $I(i) = 0$ and i has at most one black outgoing arc.

It is easy to see that in the first and second cases, we have a path $i \rightarrow^* j$ of either type A or B. This is enough to show that the lemma holds for this cases, so we continue the proof by considering only the last situation.

We can, actually, even make a stronger claim in (3): "$I(i) = 0$ and i has *exactly* one black outgoing arc". This follows from the fact the path $i \rightarrow^* j$ only has black arcs. Based on this, it is easy to see that i must have a black incoming arc. Thus, let's assume, by induction, that there is a path of either type A or B from some node i^* to i (assuming that we, temporarily, remove the black arc leaving i). By joining both paths, we get a path $i^* \rightarrow^* j$ of either type A or B and, consequently, we have proof that the lemma holds (this applies to all nodes $j \in V$ such that $I(j) = 1$). The base cases for this induction on i are precisely (1) and (2) – see above. Given the topology of G, we know that node m will always ensure by default that one of the base cases holds. Even if $I(m) = 0$, m must have at least two black outgoing arcs, because it only has one red incoming arc (from node α). □

Finally, we are ready to prove theorem 5.

Theorem 5. *Given an optimal schedule S' of jobs J', there always exists an optimal schedule S of the jobs $J' \cup \{k\}$ (with rank$(k) \geq \max_{j \in J'}\{\text{rank}(j)\}$) that does not re-schedule jobs of size 2 in S'. That is, for all jobs $j \in J'$:*

$$j \in S'_y \implies j \in S_y, \quad \textit{if} \;\; p_j = 2$$

Proof. Let $G = (V, A, k, \hat{m}, S')$ be any optimal re-schedule graph that is able to construct S. We will prove that the theorem holds by transforming G into a new re-schedule graph G^* that doesn't have any red arcs, except the one leaving node α.

We can assume, given lemma 8, that G is a simplified graph. In addition, we consider that for every node $i \in V$ we have that $I(i) \leq 1$. If not, we simply drop all re-schedules of G and directly schedule job k in a machine v such that $I(v) > 1$. The resulting graph $G^* = (\{\alpha, v\}, \{(\alpha, v)_k)\}, k, v, S')$ still produces an optimal and feasible schedule, because $\text{rank}(k) \geq \text{rank}(i)$, $\forall i \in V$.

Since $p_k = 2$, there must exist at least two distinct nodes i, j in V such that $I(i) = I(j) = 1$. By lemma 9, we know that there are two paths $P_i = m \rightarrow^* i$ and $P_j = m' \rightarrow^* j$ of types A or B. These paths are, necessarily, different, because $i \neq j$. Now, assume w.l.o.g that $\text{rank}(m) \geq \text{rank}(m')$. Since job k can be scheduled on any currently loaded machine, let's give it to m and forget all re-schedules of G.

If **(i)** P_i is of type A, we can re-schedule an extra job of size 1 from m to m' and keep the re-schedules given by paths P_i and P_j. Otherwise, **(ii)** if P_i is of type B, we simply change the black outgoing arc of m to enter node m' instead. Additionally, we follow the re-schedules given by P_j. In both cases, we built a re-schedule graph where no job of size 2 is re-scheduled and no machine changes its makespan (relatively to S'). Furthermore, the solutions are feasible, because we only introduced one new feasible arc $(m, m')_v$. Feasibility comes from the fact that $\text{rank}(v) \geq \text{rank}(m) \geq \text{rank}(m')$).

It can be easily checked that the previous argument is still valid when $m = m'$ or when at least one of the paths P_i or P_j are empty (see the example in fig. 4.10).

Formally, we define the resulting re-schedule graph as $G^* = (V^*, A^*, k, m, S')$, where:

$$
\begin{aligned}
\boldsymbol{V^*} &= \mathit{nodes}(P_i) \cup \mathit{nodes}(P_j) \cup \{\alpha\} \qquad \text{and} \\
\boldsymbol{A^*} &= \mathit{arcs}(P_j) \cup \{(\alpha, m)_k\} \\
&\quad \cup \begin{cases} \mathit{arcs}(P_i), & \text{if } P_i \text{ is of type } A \\ \varnothing, & \text{otherwise.} \end{cases} \\
&\quad \cup \begin{cases} \{(m, m')_v\}, & \text{if } m \neq m' \\ \varnothing, & \text{otherwise.} \end{cases}
\end{aligned}
$$

In this representation, $\mathit{nodes}(P)$ ($\mathit{arcs}(P)$) is the set of all nodes (arcs) in the path P. The arc $(m, m')_v$ is only added if $m \neq m'$, in order to avoid unnecessary loops. Job v is the one whose corresponding arc a in G leaves m and $a \notin \mathit{arcs}(P_i)$. □

In figures 4.9 and 4.10, we can see examples illustrating the argument for the proof of theorem 5.

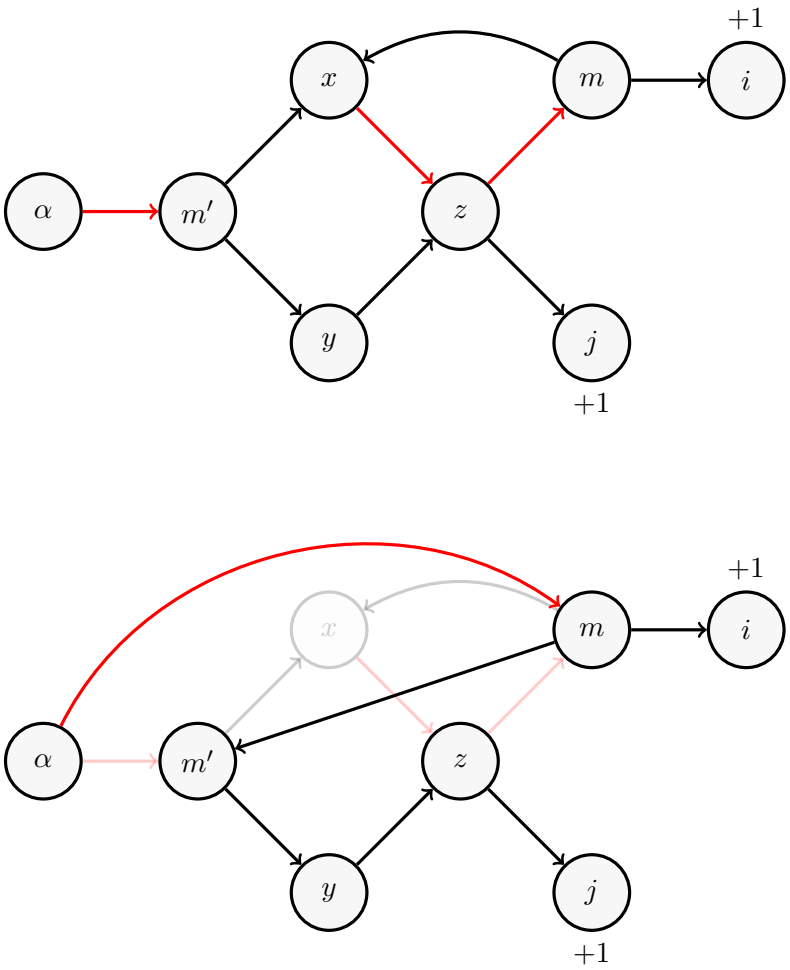


Figure 4.9: Example of the construction of a re-schedule graph (on bottom) with only one red arc, given the graph on the top. Nodes labelled +1 correspond to machines that increase their makespan by 1 unit. We assume that $\text{rank}(m) \geq \text{rank}(m')$. The technique used is described in the proof of theorem 5.

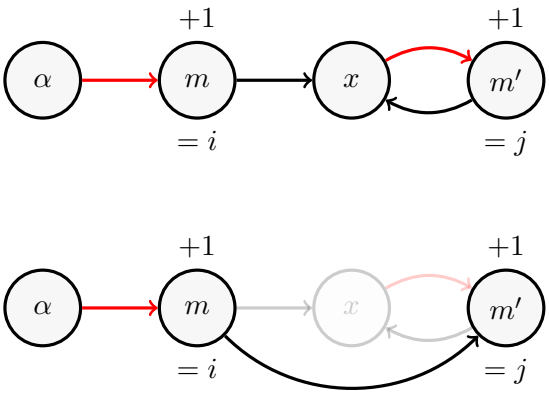


Figure 4.10: Example of the construction of a re-schedule graph (on bottom) with only one red arc, given the graph on the top. Nodes labelled +1 correspond to machines that increase their makespan by 1 unit. We assume that $\text{rank}(m) \geq \text{rank}(m')$. The technique used is described in the proof of theorem 5. Note that, in this case, we have that $m = i$ and $m' = j$.

4.3 Generalizing to nested machine sets

In this section we will try to extend the solution discussed in section 4.2 to a more general case of the *scheduling with job assignment restrictions* problem (equivalently, $R \mid p_{ij} \in \{p_j, \infty\} \mid C_{max}$) introduced by Muratore, Schwarz and Woeginger [29] as mentioned in section 4.1. Instead of assuming that the machine sets are ordered by inclusion, we will consider the case where the machine sets $\mathcal{M}(j)$ are *nested*, i.e., for any two jobs $i, j \in J$:

$$\begin{array}{lll} \text{either} & \mathcal{M}(i) \cap \mathcal{M}(j) = \varnothing, & \text{or} \\ & \mathcal{M}(i) \subseteq \mathcal{M}(j), & \text{or} \\ & \mathcal{M}(i) \supseteq \mathcal{M}(j). & \end{array}$$

Once again, notice the correspondence to the special case of the non-preemptive speed scaling problem ($S \mid r_j, d_j \mid E$), when the job lifetimes form a laminar structure (see section 1.4.2). Figure 1.3, on page 7, illustrates how this instance generalises the special case of ordered machine sets. Notice, however, that the horizontal axis represents time in the speed scaling setting, instead of machines.

4.3.1 Preliminaries

Note that the key to the solution of the problem with ordered machine sets is the following strategy:

> (S) For every two jobs i, j in J, such that $\mathcal{M}(i) \subseteq \mathcal{M}(j)$, the first attempt of scheduling i must be *before* the first attempt of scheduling j (recall that there might be re-schedules). Moreover, if $p_i = p_j = 2$ then i is guaranteed to be scheduled before j (jobs of size 2 are never re-scheduled).

Intuitively, we should be able to apply a similar strategy in the case of nested machine sets, since its structure has also the potential for establishing such an order on the jobs/machines. In the case of ordered machine sets, this ordering was easily accomplished by the notion of ranking (of jobs and machines): recall that $\text{rank}(j)$ grows with increasing values of $|\mathcal{M}(j)|$ (see definition 5). However, when machine sets are nested, it might be the case that $\mathcal{M}(i) \cap \mathcal{M}(j) = \varnothing$ for some jobs i and j. Hence, we can't have a total order, but only a *partial* one.

To that end, we will consider, from now on, that jobs and machines are elements of *posets*.

Definition 12 (Poset)**.** A *poset*, or partially ordered set, $(P, \leq)$ is a set P equipped with a (non-strict) partial-order relation $\leq$, i.e., a binary relation that satisfies, for all a, b and c in P the following properties:

- $a \leq a$ (*reflexivity*)
- if $a \leq b$ and $b \leq a$ then $a = b$ (*antisymmetry*)
- if $a \leq b$ and $b \leq c$ then $a \leq c$ (*transitivity*)

Thus, let $M^\bullet = (M, \leq_M)$ be the poset of all machines and $J^\bullet = (J, \leq_J)$ be the poset of all jobs, where the partial-order is defined as follows:

(for any two jobs i and j in $J^\bullet$)	(for any two machines i and j in $M^\bullet$)
$i \leq_{\boldsymbol{J}} j \iff \mathcal{M}(i) \subseteq \mathcal{M}(j)$	$i \leq_{\boldsymbol{M}} j \iff \mathcal{M}'(i) \subseteq \mathcal{M}'(j)$

Recall that $\mathcal{M}'(m)$ is the smallest set of machines (in F) that contains m (see section 4.1.1).

For simplicity, we will use the operator $\leq$ for both jobs and machines. Its meaning can be determined from the context. We say that a job $i \in J^\bullet$ is *greater* than a job $j \in J^\bullet$ if and only if $i \leq j$. Naturally, the same reasoning can be done about machines.

Definition 13 (Maximal and minimal elements)**.** Let $(P, \leq)$ be a poset and $P' \subseteq P$. A *maximal* (resp. *minimal*) element of P' is an element of P' that is not smaller (resp. greater) than any other element of P'.

4.3.2 The solution

This problem can be solved in the same manner as the problem where machine sets $\mathcal{M}(j)$ are totally ordered by inclusion (see section 4.2). We also iterate over the set of jobs, scheduling one of them in each iteration. But this time, we follow any *topological ordering* associated with the job precedences, given by poset $J^\bullet$. This way, we can handle subsequent minimal jobs of J and, therefore, keep strategy (S) – previously mentioned. Note that there can be more than one possible topological orderings, but none of them prevails over another. Once again, if the job being scheduled has size 1 or if it can be scheduled in a machine without increasing the maximum makespan, then we schedule it on any machine of minimum makespan. Otherwise, we need to call, again, the re-schedule routine.

Throughout the rest of this section, differences between the new algorithm $\mathcal{A}'$ and the previous one, $\mathcal{A}$, will be highlighted in blue. Common statements remain expressed in black.

Below, we can see that, in the main loop of the algorithm, we only take care of the order along which jobs and machines are handled.

Algorithm $\mathcal{A}'$

1. Let S be an empty schedule

2. Let $J' \leftarrow J$

3. WHILE $J \neq \varnothing$, DO

- **(a)** Pick any minimal job $k \in J'$
- **(b)** Pick any minimal machine $m \in \mathcal{M}(k)$
- **(c)** IF $p_k = 1$ or $\gamma(m) + p_k \leq \Gamma(S)$, THEN
  - **i.** Schedule, under S, job k in machine m
- **(d)** ELSE
  - **i.** $S^{new} \leftarrow \arg\min_{i \in \mathcal{M}(k)} \{ \Gamma(\textbf{Re-schedule'}(S, k, i)) \}$
  - **ii.** $S \leftarrow S^{new}$
- **(e)** $J' \leftarrow J' \setminus \{k\}$

4. RETURN S

Naturally, the routine responsible for re-scheduling jobs needs some modifications as well, but the essence of the function is the same. Once more, it is called for every machine $i \in \mathcal{M}(k)$ and it starts by scheduling job k in machine i, from which it is removed a set T of jobs. The method for determining this set doesn't required any modifications.

Procedure Compute' T

1. Let $T \leftarrow \{\}$

2. Let γ' be i's makespan before scheduling job k on it

3. WHILE $\gamma(i) > \gamma'$ and i has jobs of size 1, DO

  (a) Let j be any job of size 1 scheduled in i

  (b) Unschedule j from machine i

  (c) $T \leftarrow T \cup \{j\}$

We proceed by determining the set of jobs T^*. By analogy with the previous alg. $\mathcal{A}$, this set contains the greatest possible jobs that can be obtained by successive job exchanges (see previous section 4.2).

Procedure Compute' T^*

(**Compute'** T)

1. Let $T^* \leftarrow \{\}$

2. Let $M' \leftarrow \mathcal{M}(k)$

3. WHILE $M' \neq \varnothing$ and $T \neq \varnothing$, DO

  (a) Remove any minimal machine m' from M'

  (b) Move every job j from T to T^*, if $m' \notin \mathcal{M}(j)$

  (c) **Update'** T

4. $T^* \leftarrow T^* \cup T$

As we can see in the above procedure, no major change was done.

Obviously, it is also necessary to modify the routine Update T, so as to reflect the precedence of jobs given by poset $J^\bullet$.

Procedure Update' T

1. Let $T_{new} \leftarrow \{\}$ and $R \leftarrow T \cup \{\text{jobs of size 1 scheduled in } m'\}$
2. REPEAT $|T|$ times:
   (a) Pick any maximal job $r \in R$
   (b) Move r from R to T_{new}
3. Schedule all jobs $T \setminus T_{new}$ in m'
4. Unschedule all jobs $T_{new} \setminus T$ from m'
5. $T \leftarrow T_{new}$

Finally, we can collect all the updates made into the new **Re-schedule'** function and verify that indeed no major change is necessary. In fact, the overall workflow of this routine is exactly the same as the one described in section 4.2.

Function Re-schedule'

Input: schedule S, job k, machine i
Output: schedule S (modified)

1. Update S by scheduling job k on machine i
2. **Compute' T**
3. **Compute' T^***
4. Update S by scheduling all jobs of T^* using the strategy of the main algorithm
5. RETURN S

Only local adjustments were carried out in order to account for the new precedence of jobs and machines given by the posets $J^\bullet$ and $M^\bullet$, respectively. Observe that also step 4. of Re-schedule' needs to be adapted accordingly, i.e.: by sequentially assigning *greater* jobs to machines of minimum makespan.

4.3.3 Correctness of algorithm $\mathcal{A}'$

As it can be verified in the last section, the differences between algorithms $\mathcal{A}$ and $\mathcal{A}'$ are only "superficial" ones. By that reason, showing correctness of $\mathcal{A}'$ is straightforward. In fact, it is almost just necessary to modify the proof of correctness of $\mathcal{A}$ by turning expressions referring to "ranking" (of both jobs and machines) into equivalent ones suiting the new precedence notion of jobs and machines.

Nevertheless, the rest of this section will revisit, step by step, the overall proof of correctness of $\mathcal{A}$ and point out the adjustments that are necessary to generalise it.

Recall that we started off by showing that the algorithm terminates.

Claim 1′. *Algorithm $\mathcal{A}'$ terminates after $|J|$ iterations of its main loop.*

Proof. (Equivalent to the proof of claim 1 in section 4.2.1.) □

Looking back, we based ourselves in a proof by induction on the i^{th} iteration of the algorithm. Naturally, we do the same for $\mathcal{A}'$ and, once again, we skip the correctness of the inductive step.

Theorem 3′. *Algorithm $\mathcal{A}'$ builds an optimal schedule to problem $R \mid p_{ij} \in \{p_j, \infty\},\ p_j \in \{1, 2\} \mid C_{max}$ using nested machine sets.*

Proof. (Equivalent to the proof of theorem 3, except that the inductive step is shown to be correct by sections 4.3.3.1 and 4.3.3.2. These prove, respectively, that steps 3.(c) and 3.(d) build optimal schedules for all jobs considered so far.) □

4.3.3.1 Correctness of $\mathcal{A}'$'s step 3.(c)

As we can verify, the proof of correctness of the equivalent step in alg. $\mathcal{A}$ (see section 4.2.1.1) doesn't mention, in any part, ranks of jobs or machines. Thus, we can directly reuse it for this case, with no need for adjustments. In that proof, we exploited the fact that this step (or its corresponding step in alg. $\mathcal{A}$) is only executed when job k has size 1 or when it is possible to directly schedule k without changing the current maximum makespan. For the latter case, we showed that the resulting schedule is optimal in lemma 3. For the former, lemma 4 was used instead. Below, we can review both lemma statements.

Lemma 3. *Let S' be an optimal schedule of jobs J' and S a schedule of jobs $J' \cup \{j\}$. If $\Gamma(S') = \Gamma(S)$, then S is also optimal.*

Lemma 4. *Let S' be a schedule of jobs J' over a set of machines M'. If, for every two machines i, j in M', we have that $|\gamma(i) - \gamma(j)| \leq 1$, then S' is an optimal schedule of jobs J'. The converse is not true.*

4.3.3.2 Correctness of $\mathcal{A}'$'s step 3.(d)

The proof of correctness of this step is a bit more complex (compared to step 3.(c)), but it will also rely on the proof that the corresponding step in alg. $\mathcal{A}$ is correct. In total, we used one theorem, one definition and three lemmas, before we could prove the correctness of the step in theorem 6.

We began by assuming that: there is a solution for scheduling k that does not require re-assigning jobs of size 2 to different machines, and keeps scheduled all the jobs considered in previous iterations. Hence, we do the same now.

Theorem 5′. *Given an optimal schedule S' of jobs J', there always exists an optimal schedule S of the jobs $J' \cup \{k\}$ (with $k \geq \max J'$) that does not re-schedule jobs of size 2 in S'. That is, for all jobs $j \in J'$:*

$$j \in S'_y \implies j \in S_y, \quad \text{if } p_j = 2$$

The proof for the above theorem is based on the proof of theorem 5, but a more detailed explanation is given in section 4.3.4.

After, we assumed that the output solution of alg. $\mathcal{A}$, denoted by S, isn't optimal, but some schedule S^* is. The overall idea was to prove, by contradiction, the optimality of S. To that end, we partitioned the set of machines $\mathcal{M}(k)$ into 3 subsets M_1, M_2 and M_3 according to table 4.2. After, we formalised a mechanism for transforming S into S^* (see below) that would simplify the rest of the proof.

Definition 7 (Re-schedule sequence)**.** A re-schedule sequence (π_M, π_J, l) is a series of l feasible re-schedules defined by a permutation $\pi_M : \{1, \ldots, l+1\} \to M$ of machines and a permutation $\pi_J : \{1, \ldots, l\} \to J$ of jobs such that job $\pi_J(i)$ is re-scheduled from machine $\pi_M(i)$ to machine $\pi_M(i+1)$, for all $1 \leq i \leq l$ and as long as $\pi_M(i+1) \in \mathcal{M}(\pi_J(i))$.

The upcoming lemmas would be the final step before concluding the proof with theorem 6. Below, we can recall them. Except to the last one, all three lemmas can be proved in exactly the same way as they were proved for ordered machine sets.

Lemma 5. *If S is not optimal we need at least one re-schedule sequence (π_M, π_J, l) where $\pi_M(1) \in M_1$ and $\pi_M(l+1) \in M_3$ in order to be able to transform S into S^*.*

Proof. (See proof of lemma 5 on section 4.2.1.2.) □

Lemma 6. *Every machine in M_1 schedules (under S) at least one job from the set T^* if $T^* \neq \varnothing$.*

Proof. (See proof of lemma 6 on section 4.2.1.2.) □

Lemma 7′. *For every job $t \in T^*$, there is no other job scheduled in some machine of $\mathcal{M}(t)$ that is greater than t.*

Proof. In the original proof of lemma 7, we could argue that $\text{rank}(t) \leq \text{rank}(k)$ for all jobs $t \in T^*$, given that the machine sets are totally ordered. Unfortunately, we can no longer use the same argument to sustain the analogous statement $t \leq k$, for all jobs $t \in T^*$. Hence, we make the necessary adjustments in the proof.

Once again, if there was a job t' such that $t' > t$ for some job t in T^* and t' is scheduled in some machine x of $\mathcal{M}(t)$, either:

(i) the algorithm failed to select the maximal jobs in step 2.(a) of the routine **Update'** T;
(ii) or: $x \notin \mathcal{M}(k)$, the set of machines covered by the procedure **Compute'** T^*.

Assuming that the algorithm runs according to its description, only the latter case (ii) might disproof the lemma. It is easy to see (from the description of the procedure **Compute'** T^*) that all jobs in T^* were previously scheduled in some machine of $\mathcal{M}(k)$, which implies that the machine sets associated to these jobs are nested in $\mathcal{M}(k)$. In other words: $\mathcal{M}(t) \subseteq \mathcal{M}(k)$ for all jobs t in T^*. However, if $x \notin \mathcal{M}(k)$ (ii), then it must be the case that $\mathcal{M}(t) \not\subset \mathcal{M}(k)$, a contradiction. □

Note that the proof above is very similar to the original proof of lemma 7, but, for ease of understanding and recall, the common parts of both proofs were repeated.

Finally, we were able to prove the correctness of step 2.(d) of alg. $\mathcal{A}$ by applying the last lemmas in the proof of theorem 6. Thus, we adopt one last time the same strategy for the case of nested machine sets.

Theorem 6′. *Step 3.(d) of algorithm $\mathcal{A}$ builds an optimal schedule for job k and all the previously scheduled jobs.*

Proof. (Straightforward generalisation of the original proof of theorem 6: replace any expression comparing jobs, or machines, by their rank into equivalent ones that use the new partial-order relation defined by the posets $J^\bullet$ and $M^\bullet$.) □

4.3.4 Proof of theorem 5′

Theorem 5′. *Given an optimal schedule S' of jobs J', there always exists an optimal schedule S of the jobs $J' \cup \{k\}$ (with $k \geq \max J'$) that does not re-schedule jobs of size 2 in S'. That is, for all jobs $j \in J'$:*

$$j \in S'_y \implies j \in S_y, \quad \textit{if} \;\; p_j = 2$$

Proof. The proof follows, trivially, from a generalisation of the proof of theorem 5, since at no point of the demonstration we benefit explicitly from the fact that the machine sets are ordered. Instead, we only take advantage of the fact that job k has the highest rank and can, therefore, be assigned to any machine currently scheduling jobs. Moreover, we only account for this fact in the end of section 4.2.3, on, precisely, the proof of theorem 5. Nonetheless, we need to make a few adjustments so as to adapt the argument to nested machine sets.

By looking at section 4.2.3, we can verify that, before the conclusive proof of theorem 5, it is described (in a series of definitions and lemmas) a process for transforming any schedule into an optimal one. This is accomplished by *re-schedule graphs*[(1)] that are, gradually, modified until they satisfy a set of properties[(2)], which in turn, simplify its final transformation into a re-schedule graph that does not re-schedule jobs of size 2[(3)].

In order to be able to re-use the proof of theorem 5, we need to make sure that job k can be scheduled in any machine that is part of a suitable re-schedule graph. This way, we have more freedom to manipulate the graph while satisfying the feasibility constraints. Thus, let us update the definition of a simplified re-schedule graph.

Definition 11′ (Simplified graph)**.** A *simplified graph* $G = (V, A, k, m, S)$ is a re-schedule graph with the following properties:

1. node α is the only source node in G;
2. G does not have unicoloured cycles, i.e., cycles whose arcs have the same colour;
3. G does not have any path of type C;
4. all machines of G are elements of $\mathcal{M}(k)$, i.e., $V \setminus \{\alpha\} \subseteq \mathcal{M}(k)$.

The only change we made in the above definition was adding property 4, which is implicitly present in all re-schedule graphs if the machine sets are ordered.

As a consequence of redefining simplified graphs, we need to also update the proof of lemma 8.

(1) See definition 8.
(2) See definition 11 and lemma 8.
(3) See lemma 9 and theorem 5.

Lemma 8′. *Given an optimal schedule S' of jobs J', there always exists a simplified graph whose activation transforms it into an optimal schedule S of the jobs $J' \cup \{k\}$.*

Proof. Let G be any re-schedule graph able to transform S' into S. In the previous version of this proof (for ordered machine sets), we can verify that such graph always exists and can be modified to meet properties 1, 2 and 3. Given that these properties are somewhat independent of each other, we can afford to change G and let it reflect each property, individually.

• *Property 4.* Remove all nodes u in V such that $u \notin \mathcal{M}(k)$. Moreover, remove all incoming and outgoing arcs of u. Let G be the resulting graph.

Note that after this transformation, G still holds a feasible and optimal schedule. The reason is that no new arc was added and G was, initially, feasible. Moreover, the increments of all removed nodes remained null and no other node in G changed its increment. Hence, G holds an optimal schedule by claim 2 (see section 4.2.3), using G' as the graph before all transformations. The reason why no other machine changed its increment (besides the ones removed from G) is the fact that none of the removed arcs has an endpoint in any of the machines of $\mathcal{M}(k)$. The only way that such an arc could exist, would be if we had, previously, scheduled a job j such that $j > k$ (since it must be the case that $\mathcal{M}(k) \subset \mathcal{M}(j)$). But, this, clearly, never happens under the policy followed by algorithm $\mathcal{A}'$. □

Having concluded the previous proof, we are now in a better position of generalising theorem 5. As a matter of fact, all we need to do is change all expressions comparing jobs (and machines) by their rank into equivalent expressions that use the new partial-order associated to jobs (and machines). It can be easily verified that this adjustment does not violate any feasibility constraint, given the new definition of simplified graphs.

□

Chapter 5

Conclusion

This thesis considered a variant of the speed scaling problem, in which processors aren't allowed to interrupt (and later resume) the execution of jobs. Although this setting is much harder than the preemptive version (which is in P), the set of feasible solutions to the former problem is significantly smaller.

The initial goal of this thesis was to improve the approximation ratio of the non-preemptive speed scaling problem, using only combinatorial algorithms. Sadly, this task turned out to be too difficult to achieve within the available time. Nevertheless, several contributions were made to special instances of the problem. One of the main results is a substantial reduction on the time complexity for the case that all jobs share equal work volumes and only one processor is used to schedule them. This improvement derived from a better analysis of the polynomial-time solutions given so far, which allowed us to take full advantage of their potential. In addition, by a simple extension of one of these solutions, it was possible to demonstrate that the problem remains in P when the number of jobs of unrestricted work volumes is limited to any constant. Although straightforward, this contribution is one step further to better understand the complexity class of the non-preemptive speed scaling problem, whose general instance is known to be strongly NP-hard, if using a single processor. The last major result was the design of an independent algorithm that solves, optimally, a special instance of the problem *scheduling with job assignment restrictions* [29]. An elaborate proof of correctness is also given in the same chapter. Although for a special case (all jobs have a processing time of either one or two), this algorithm is optimal and it improves the time complexity of the PTAS (see [29]) that is able to solve instances with unrestricted processing times. Both approaches assume, however, that jobs have nested assignment restrictions (see section 4.1). The reason for considering a different scheduling problem is related to the connection between this and the speed scaling problem, which was explored by Huang and Ott [22] to design a QPTAS for laminar instances.

Besides the improvements mentioned in the last paragraph, two less significant con-

tributions (see section 3.3) were also made, with respect to other special instances of the non-preemptive speed scaling problem. It is important to mention that both of them relied on the well known polynomial-time algorithm given by Huang and Ott [22] for the case of equal work volumes. One of the improvements consisted in decreasing the time complexity when all jobs define a purely laminar structure. The other one considered extending this result to the context of multiple processors and it showed that, if the number of processors is a constant, the problem is still in P.

Further work. It is very likely that the strategy adopted in chapter 4 to solve a special instance of the problem *scheduling with job assignment restrictions* can be generalised to instances that include any two processing times (instead of only the values one and two). Moreover, I believe that this method can be further extended to solve, in polynomial-time, instances where the number of different processing times is a constant. Ultimately, it could even give rise to an approximation algorithm that is able to handle unrestricted processing times. It would be interesting to then compare the algorithm performance and its approximation ratio with the current best ones. Unfortunately, due to lack of time, none of these ideas was explored. For the same reason, the techniques used in the same chapter weren't further employed in the original speed scaling problem – goal that, originally, motivated the study of a different scheduling problem.

Bibliography

[1] *2014 Key World Energy Statistics" (PDF).* IEA. 2014. URL: http://www.iea.org/publications/freepublications/.

[2] Susanne Albers, Antonios Antoniadis, and Gero Greiner. "On multi-processor speed scaling with migration: extended abstract". In: *SPAA 2011: Proceedings of the 23rd Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Jose, CA, USA, June 4-6, 2011 (Co-located with FCRC 2011).* 2011, pp. 279–288. DOI: 10.1145/1989493.1989539. URL: http://doi.acm.org/10.1145/1989493.1989539.

[3] Susanne Albers, Fabian Müller, and Swen Schmelzer. "Speed scaling on parallel processors". In: *SPAA 2007: Proceedings of the 19th Annual ACM Symposium on Parallelism in Algorithms and Architectures, San Diego, California, USA, June 9-11, 2007.* 2007, pp. 289–298. DOI: 10.1145/1248377.1248424. URL: http://doi.acm.org/10.1145/1248377.1248424.

[4] Eric Angel, Evripidis Bampis, and Vincent Chau. "Throughput Maximization in the Speed-Scaling Setting". In: *31st International Symposium on Theoretical Aspects of Computer Science (STACS 2014), STACS 2014, March 5-8, 2014, Lyon, France.* 2014, pp. 53–62. DOI: 10.4230/LIPIcs.STACS.2014.53. URL: http://dx.doi.org/10.4230/LIPIcs.STACS.2014.53.

[5] Eric Angel et al. "Speed Scaling on Parallel Processors with Migration". In: *Euro-Par 2012 Parallel Processing - 18th International Conference, Euro-Par 2012, Rhodes Island, Greece, August 27-31, 2012. Proceedings.* 2012, pp. 128–140. DOI: 10.1007/978-3-642-32820-6_15. URL: http://dx.doi.org/10.1007/978-3-642-32820-6_15.

[6] Antonios Antoniadis and Chien-Chung Huang. "Non-preemptive Speed Scaling". In: *Algorithm Theory - SWAT 2012 - 13th Scandinavian Symposium and Workshops, Helsinki, Finland, July 4-6, 2012. Proceedings.* 2012, pp. 249–260. DOI: 10.1007/978-3-642-31155-0_22. URL: http://dx.doi.org/10.1007/978-3-642-31155-0_22.

[7] Antonios Antoniadis, Chien-Chung Huang, and Sebastian Ott. "A Fully Polynomial-Time Approximation Scheme for Speed Scaling with Sleep State". In: *Proceedings of the Twenty-Sixth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA*

2015, San Diego, CA, USA, January 4-6, 2015. 2015, pp. 1102–1113. DOI: 10.1137/1.9781611973730.74. URL: http://dx.doi.org/10.1137/1.9781611973730.74.

[8] Evripidis Bampis, Dimitrios Letsios, and Giorgio Lucarelli. "Green Scheduling, Flows and Matchings". In: *Algorithms and Computation - 23rd International Symposium, ISAAC 2012, Taipei, Taiwan, December 19-21, 2012. Proceedings*. 2012, pp. 106–115. DOI: 10.1007/978-3-642-35261-4_14. URL: http://dx.doi.org/10.1007/978-3-642-35261-4_14.

[9] Evripidis Bampis, Dimitrios Letsios, and Giorgio Lucarelli. "Speed-Scaling with No Preemptions". In: *Algorithms and Computation - 25th International Symposium, ISAAC 2014, Jeonju, Korea, December 15-17, 2014, Proceedings*. 2014, pp. 259–269. DOI: 10.1007/978-3-319-13075-0_21. URL: http://dx.doi.org/10.1007/978-3-319-13075-0_21.

[10] Evripidis Bampis et al. "Energy Efficient Scheduling and Routing via Randomized Rounding". In: *IARCS Annual Conference on Foundations of Software Technology and Theoretical Computer Science, FSTTCS 2013, December 12-14, 2013, Guwahati, India*. 2013, pp. 449–460. DOI: 10.4230/LIPIcs.FSTTCS.2013.449. URL: http://dx.doi.org/10.4230/LIPIcs.FSTTCS.2013.449.

[11] Evripidis Bampis et al. "From Preemptive to Non-preemptive Speed-Scaling Scheduling". In: *Computing and Combinatorics, 19th International Conference, COCOON 2013, Hangzhou, China, June 21-23, 2013. Proceedings*. 2013, pp. 134–146. DOI: 10.1007/978-3-642-38768-5_14. URL: http://dx.doi.org/10.1007/978-3-642-38768-5_14.

[12] Nikhil Bansal, Tracy Kimbrel, and Kirk Pruhs. "Dynamic Speed Scaling to Manage Energy and Temperature". In: *45th Symposium on Foundations of Computer Science (FOCS 2004), 17-19 October 2004, Rome, Italy, Proceedings*. 2004, pp. 520–529. DOI: 10.1109/FOCS.2004.24. URL: http://doi.ieeecomputersociety.org/10.1109/FOCS.2004.24.

[13] Brad D. Bingham and Mark R. Greenstreet. "Energy Optimal Scheduling on Multiprocessors with Migration". In: *IEEE International Symposium on Parallel and Distributed Processing with Applications, ISPA 2008, Sydney, NSW, Australia, December 10-12, 2008*. 2008, pp. 153–161. DOI: 10.1109/ISPA.2008.128. URL: http://dx.doi.org/10.1109/ISPA.2008.128.

[14] David M. Brooks et al. "Power-Aware Microarchitecture: Design and Modeling Challenges for Next-Generation Microprocessors". In: *IEEE Micro* 20.6 (2000), pp. 26–44. DOI: 10.1109/40.888701. URL: http://doi.ieeecomputersociety.org/10.1109/40.888701.

[15] Vincent Cohen-Addad et al. "Energy-Efficient Algorithms for Non-preemptive Speed-Scaling". In: *Approximation and Online Algorithms - 12th International Workshop, WAOA 2014, Wrocław, Poland, September 11-12, 2014, Revised Selected Papers*. 2014, pp. 107–118. DOI: 10.1007/978-3-319-18263-6_10. URL: http://dx.doi.org/10.1007/978-3-319-18263-6_10.

[16] Tomás Ebenlendr, Marek Krcál, and Jiri Sgall. "Graph balancing: a special case of scheduling unrelated parallel machines". In: *Proceedings of the Nineteenth Annual ACM-SIAM Symposium on Discrete Algorithms, SODA 2008, San Francisco, California, USA, January 20-22, 2008*. 2008, pp. 483–490. URL: http://dl.acm.org/citation.cfm?id=1347082.1347135.

[17] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN: 0-7167-1044-7.

[18] Celia A. Glass and Hans Kellerer. "Parallel machine scheduling with job assignment restrictions". In: *Naval Research Logistics (NRL)* 54.3 (2007), pp. 250–257. ISSN: 1520-6750. DOI: 10.1002/nav.20202. URL: http://dx.doi.org/10.1002/nav.20202.

[19] Celia A. Glass and H. R. Mills. "Scheduling unit length jobs with parallel nested machine processing set restrictions". In: *Computers & OR* 33 (2006), pp. 620–638. DOI: 10.1016/j.cor.2004.07.010. URL: http://dx.doi.org/10.1016/j.cor.2004.07.010.

[20] Kinshuk Govil, Edwin Chan, and Hal Wasserman. "Comparing Algorithm for Dynamic Speed-setting of a Low-power CPU". In: *Proceedings of the 1st Annual International Conference on Mobile Computing and Networking*. MobiCom '95. Berkeley, California, USA: ACM, 1995, pp. 13–25. ISBN: 0-89791-814-2. DOI: 10.1145/215530.215546. URL: http://doi.acm.org/10.1145/215530.215546.

[21] R.L. Graham et al. "Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey". In: *Discrete Optimization IIProceedings of the Advanced Research Institute on Discrete Optimization and Systems Applications of the Systems Science Panel of NATO and of the Discrete Optimization Symposium co-sponsored by IBM Canada and SIAM Banff, Aha. and Vancouver*. Ed. by E.L. Johnson P.L. Hammer and B.H. Korte. Vol. 5. Annals of Discrete Mathematics. Elsevier, 1979, pp. 287–326. DOI: http://dx.doi.org/10.1016/S0167-5060(08)70356-X. URL: http://www.sciencedirect.com/science/article/pii/S016750600870356X.

[22] Chien-Chung Huang and Sebastian Ott. "New Results for Non-Preemptive Speed Scaling". In: *Mathematical Foundations of Computer Science 2014 - 39th International Symposium, MFCS 2014, Budapest, Hungary, August 25-29, 2014. Proceedings, Part II*. 2014, pp. 360–371. DOI: 10.1007/978-3-662-44465-8_31. URL: http://dx.doi.org/10.1007/978-3-662-44465-8_31.

[23] Hark-Chin Hwang, Soo Y. Chang, and Kangbok Lee. "Parallel machine scheduling under a grade of service provision". In: *Computers & OR* 31.12 (2004), pp. 2055–2061. DOI: 10.1016/S0305-0548(03)00164-3. URL: http://dx.doi.org/10.1016/S0305-0548(03)00164-3.

[24] Sandy Irani, Sandeep K. Shukla, and Rajesh K. Gupta. "Algorithms for power savings". In: *Proceedings of the Fourteenth Annual ACM-SIAM Symposium on Discrete Algorithms, January 12-14, 2003, Baltimore, Maryland, USA.* 2003, pp. 37–46. URL: http://dl.acm.org/citation.cfm?id=644108.644115.

[25] J.L.W.V. Jensen. "Sur les fonctions convexes et les inégalités entre les valeurs moyennes". French. In: *Acta Mathematica* 30.1 (1906), pp. 175–193. ISSN: 0001-5962. DOI: 10.1007/BF02418571. URL: http://dx.doi.org/10.1007/BF02418571.

[26] Harold W. Kuhn. "The Hungarian Method for the Assignment Problem". In: *Naval Research Logistics Quarterly* 2.1–2 (Mar. 1955), pp. 83–97. DOI: 10.1002/nav.3800020109.

[27] Jan Karel Lenstra, David B. Shmoys, and Éva Tardos. "Approximation Algorithms for Scheduling Unrelated Parallel Machines". In: *28th Annual Symposium on Foundations of Computer Science, Los Angeles, California, USA, 27-29 October 1987.* 1987, pp. 217–224. DOI: 10.1109/SFCS.1987.8. URL: http://dx.doi.org/10.1109/SFCS.1987.8.

[28] Minming Li, Becky Jie Liu, and Frances F. Yao. "Min-Energy Voltage Allocation for Tree-Structured Tasks". In: *Computing and Combinatorics, 11th Annual International Conference, COCOON 2005, Kunming, China, August 16-29, 2005, Proceedings.* 2005, pp. 283–296. DOI: 10.1007/11533719_30. URL: http://dx.doi.org/10.1007/11533719_30.

[29] Gabriella Muratore, Ulrich M. Schwarz, and Gerhard J. Woeginger. "Parallel machine scheduling with nested job assignment restrictions". In: *Oper. Res. Lett.* 38.1 (2010), pp. 47–50. DOI: 10.1016/j.orl.2009.09.010. URL: http://dx.doi.org/10.1016/j.orl.2009.09.010.

[30] Daniel Dominic Sleator and Robert Endre Tarjan. "Amortized Efficiency of List Update and Paging Rules". In: *Commun. ACM* 28.2 (1985), pp. 202–208. DOI: 10.1145/2786.2793. URL: http://doi.acm.org/10.1145/2786.2793.

[31] *Statistical Review of World Energy 2015.* BP. 2015. URL: http://www.bp.com/statisticalreview.

[32] Éva Tardos. "A strongly polynomial minimum cost circulation algorithm". In: *Combinatorica* 5.3 (1985), pp. 247–256. DOI: 10.1007/BF02579369. URL: http://dx.doi.org/10.1007/BF02579369.

[33] Mark Weiser et al. “Scheduling for Reduced CPU Energy”. In: *Proceedings of the First USENIX Symposium on Operating Systems Design and Implementation (OSDI), Monterey, California, USA, November 14-17, 1994*. 1994, pp. 13–23. URL: http://dl.acm.org/citation.cfm?id=1267640.

[34] Adam Wierman, Lachlan L. H. Andrew, and Ao Tang. “Power-aware speed scaling in processor sharing systems: Optimality and robustness”. In: *Perform. Eval.* 69.12 (2012), pp. 601–622. DOI: 10.1016/j.peva.2012.07.002. URL: http://dx.doi.org/10.1016/j.peva.2012.07.002.

[35] F. Frances Yao, Alan J. Demers, and Scott Shenker. “A Scheduling Model for Reduced CPU Energy”. In: *36th Annual Symposium on Foundations of Computer Science, Milwaukee, Wisconsin, 23-25 October 1995*. 1995, pp. 374–382. DOI: 10.1109/SFCS.1995.492493. URL: http://dx.doi.org/10.1109/SFCS.1995.492493.