Résultats de complexité et programmation par contraintes pour l'ordonnancement

Philippe Baptiste

Université de Technologie de Compiègne
Laboratoire Heudiasyc, UMR CNRS 6599

Résultats de complexité et programmation par contraintes pour l'ordonnancement

Habilitation à diriger des recherches présentée le 1er juillet 2002 par
Philippe Baptiste
devant le jury composé de

Jacques Carlier	Prof. de l'Université de Tech. de Compiègne
Philippe Chrétienne	Prof. de l'Université Paris VI (Rapporteur)
Bernard Dubuisson	Prof. de l'Université de Tech. de Compiègne
Pascal van Hentenryck	Prof. de l'Université de Brown (Rapporteur)
Jan-Karel Lenstra	Prof. de l'Université d'Eindhoven (Rapporteur)
Claude Le Pape	Directeur R&D – Coopération de solveurs, Ilog
Baruch Schieber	Senior Manager Optimization Center, IBM

Acknowledgments

I wish to thank all the people who have contributed to the completion of this work. It gives me great pleasure to mention them here.

First of all, I would like to thank Jacques Carlier and Claude Le Pape, not only for their help and their numerous advices but also for their friendship over the last eight years.

I'm especially grateful to Philippe Chrétienne, Pascal van Hentenryck and Jan–Karel Lenstra who have accepted to review this habilitation thesis.

I'm also indebted to Bernard Dubuisson and Rogelio Lozano, the directors of the CNRS-HeuDiaSyC Laboratory, for their continuous support over the years. Brenda Dietrich, Baruch Schieber and Jonhattan Lee deserve special thanks for the great time I'm having at IBM Watson's Mathematical Sciences Department.

I also wish to thank all my colleagues at CNRS, at the University of Technology of Compiègne and at IBM as well as my former colleagues at Bouygues and Ilog who have discussed with me different topics of this habilitation thesis.

I'm also indebted to the many people who have helped in previous work and in preparing this manuscript. I'm especially grateful to Claude Le Pape and Wim Nuijten, the coauthors of our book *Constraint-Based Scheduling* (Kluwer, 2001). This book is the basis of the second part of this manuscript. Special thanks to the other coauthors of papers that have served as raw material for this manuscript: Jacques Carlier, Jeet Gupta, Antoine Jouglet, Emmanuel Néron, Laurent Péridy, Eric Pinson, Baruch Schieber and Vadim Timkovsky. I also wish to thank Peter Brucker and Sigrid Knust for the many enlightening discussions we have had on scheduling theory.

Foreword

Baker [9] defines scheduling as the problem of allocating scarce resources to activities over time[1]. In this manuscript, we consider several scheduling problems with a variety of different environments. The two parts of this manuscript provide an overview of two of our major research interests. The first part (Chapters 2 to 5) is dedicated to polynomial time solutions of equal–processing–time scheduling problems. The second one (Chapters 7 to 15) deals with the application of Constraint Programming to scheduling. These two parts reflect the balance we have tried to keep between theoretical and more applied research.

Although all the topics covered in this manuscript are related to scheduling, Parts 1 and 2 are dedicated to very different aspects of scheduling. Some few results discussed in Part 1 are occasionally used in Part 2 but they are independent one from another. In the following, we briefly review the content Parts 1 and 2. More detailed introductions are provided in Sections 1 and 7 respectively.

Scheduling Equal Length Jobs

Looking through the listing of complexity results for scheduling problems [47] one can observe that the vast majority of problems whose complexity status is unknown involve equal–processing–time jobs. Hence, the state–of–the–art in studying the complexity of scheduling problems has reached the level at which approaches and techniques successfully working for equal–processing–time scheduling become especially important. The first part of this manuscript is dedicated to such problems. We have tried to present, within a common framework, our recent results in this area.

- In Chapter 1, after having recalled the Graham, Lawler, Lenstra and Rinnooy Kan Classification Scheme [113], we provide a brief description of equal–processing–time scheduling problems. We discuss the difference between unit and equal–processing–time scheduling problems.

- In Chapter 2, we restrict our study to the single machine case with release dates ($1|r_i, p_i = p|F$). We first show that when preemption is allowed and when processing times are equal, we can minimize the weighted number of late jobs in polynomial time. We have a slightly more general result for the non-preemptive case since our dynamic program can handle any "ordered" objective function. Hence we show that $\sum w_iU_i, \sum w_iC_i$ and $\sum T_i$ can be minimized in polynomial time.

- In Chapter 3, we consider the situation where m identical parallel machines are available. In the preemptive case we show that, under the hypothesis that all jobs are released at time 0, the total tardiness problem can be solved in polynomial time thanks to a simple LP formulation of the problem. We also present a dynamic program to minimize the weighted number of late jobs. In the non-preemptive case, we generalize the results obtained on a single machine and we show that, with arbitrary release dates, $\sum w_iU_i, \sum w_iC_i$ and $\sum T_i$ can be minimized in polynomial time for any fixed value of m. Finally, we study the weighted number of late jobs

[1]We have used the following *terminology*: For theoretical scheduling problems, we have "Jobs" and "Machines", and we follow the Classification Scheme of Graham, Lawler, Lenstra and Rinnooy Kan [113]. For multiprocessor problems, we follow the widely used "Tasks/Processors" terminology. Finally, to present our Constraint Programming model, we use the "Activity" and "Resource" terms and we will see how they can be used to model many theoretical scheduling problems (*e.g.*, job-shop, Resource-Constrained Project Scheduling, *etc.*).

criteria in a Unit Execution Time open-shop environment $Om|p_{ij} = 1, r_i|\sum w_i U_i$. We use a result from Brucker, Jurisch and Jurisch to transform the UET open-shop into a preemptive parallel machine problem and we show that it can be solved in polynomial time with a complex Dynamic Program.

- In Chapter 4, we study the situation where a set of n tasks, subjected to release dates, have to be scheduled on m parallel processors. We consider a particular multiprocessor scheduling environment where, for each task i, a fixed number $size_i$ of processors is required to execute the task. Yet, the processors required are not specified. We first show that $Pm|r_i, size_i, p_i = p|\sum C_i$ can be solved in polynomial time by dynamic programming. Then we study the unit execution time problem ($p_i = 1$) and we show that the minimum maximal tardiness can be computed in polynomial time when $size_i \in \{1, m\}$. In this problem, there are no more than two possible sizes, either 1 (small tasks) or m (tall tasks), and we refer to it as the "tall/small" problem.

- In Chapter 5, we study the situation where n jobs $\{J_1, \ldots, J_n\}$ have to be scheduled on a batching machine. Jobs cannot start before their release dates and all jobs of the same batch start and are completed simultaneously, *i.e.*, at the starting time (respectively at the completion time) of the batch. Several types of batching machines are studied and for each of them, we provide some new complexity results for the case where jobs have identical processing times.

Table 1 summarizes the new polynomial time algorithms, for equal-length jobs, that we have obtained in the last few years. As discussed in the introduction of Part 1, a *subset* only of these algorithms is presented in this manuscript.

SINGLE MACHINE	
$1\|pmtn, p_i = p, r_i\|\sum w_i U_i$	§2.1 (Dyn. Prog.)
$1\|p_i = p, r_i\|\sum w_i U_i$	§2.2 (Dyn. Prog.)
$1\|prec, pmtn, p_i = p, r_i\|\sum C_i$	[27]
PARALLEL MACHINES	
$Pm\|pmtn, p_i = p\|\sum w_i U_i$	§3.1 (Dyn. Prog.)
$P\|p_i = p, pmtn\|\sum T_i$	§3.1 (LP Form.)
$Pm\|p_i = p, r_i\|\{\sum w_i C_i, \sum T_i, \sum w_i U_i\}$	§3.2 (Dyn. Prog.)
$Pm\|p_i = 1, intree\|\sum C_i$	[27]
$Q\|p_i = 1, pmtn\|\sum U_i$	[27]
$P2\|pmtn, outtree, r_j, p_j = 1\|\sum C_j$	[26] (ext. for any m by Brucker *et al.* [42])
$P\|p_i = 1, chains, r_i\|L_{\max}$	[27] (LP Form.)
MULTIPROCESSOR SCHEDULING	
$P\|r_i, p_i = 1, size_i \in \{1, m\}\|L_{\max}$	§4.2 (LP Form. and Combinatorial Alg.)
$Pm\|r_i, p_i = p, size_i\|\sum C_i$	§4.1 (Dyn. Prog.)
OPEN-SHOP	
$Om\|p_{ij} = 1, r_i\|\sum w_i U_i$	§3.3 (Dyn. Prog.)
BATCHING for $F \in \{\sum w_i C_i, \sum T_i, \sum w_i U_i\}$	
$1\|p\text{-}batch, b < n, r_i, p_i = p\|F$	§5.2 (Dyn. Prog.)
$1\|s\text{-}batch, r_i, p_i = p\|F$	§5.2 (Dyn. Prog.)
$1\|p\text{-}batch, r_i\|regular\ function$	§5.1 (Dyn. Prog., Pseudo-Poly.)
$1\|s\text{-}batch\|\sum T_i$	[18] (Dyn. Prog., Pseudo-Poly.)

Table 1: "Closed" $p_i = p$ Problems

Constraint-Based Scheduling [2]

Constraint Programming is a problem-solving paradigm which establishes a neat distinction between, on the one hand, a precise definition of the constraints that define the problem to be solved and, on the other hand, the algorithms and heuristics enabling the selection and cancellation of decisions to solve the problem. The main principles of Constraint Programming are:

- the separation between deductive "constraint propagation" methods, generating additional constraints from existing constraints, and search algorithms, used to systematically explore the solution space [206];
- the "locality principle", which states that each constraint must propagate as locally as possible, independently of the existence or the non-existence of other constraints [207];
- the distinction between the logical representation of constraints and the control of their use, in accordance with the equation stated by Kowalski for logic programming: Algorithm = Logic + Control [132].

These principles have been widely applied in the area of scheduling, enabling the implementation of flexible and extensible scheduling systems. Indeed, with Constraint Programming all the specific constraints of a given problem can be represented and actually used as a guide toward a solution.

As the number of applications grew, the need emerged to reconcile the flexibility offered by Constraint Programming with the efficiency of specialized Operations Research algorithms. The first step consisted in adapting well-known Operations Research algorithms to the Constraint Programming framework. As a second step, the success of the resulting tools opened a new area of research aimed at the design and implementation of efficient algorithms embeddable in Constraint Programming tools.

In this thesis, we provide a non-exhaustive overview of the most widely used Constraint-Based Scheduling techniques. Following the principles of Constraint Programming, we have three distinct sections:

- Chapter 7 introduces the basic principles of Constraint Programming and provides a model of the constraints that are the most often encountered in scheduling problems.
- Chapters 8, 9, 10 and 11 are focused on the propagation of resource constraints which usually are responsible for the "hardness" of the scheduling problem. In accordance with the locality principle, Chapters 8 and 9 focus on the propagation of one resource constraint considered independently of any other constraint. Chapter 8 deals with a resource of capacity one (which can execute one activity at a time). Chapter 9 considers the more general case in which a resource can execute several activities in parallel. Chapter 10 provides a theoretical comparison of most of the constraint propagation techniques presented earlier. Chapter 11 considers a more recent line of research, allowing to efficiently handle optimization objectives within resource constraints.
- Chapters 12, 13 and 14 are dedicated to the resolution of several academic scheduling problems. These examples illustrate the use and the practical efficiency of the constraint propagation methods of the previous chapters. They also show that besides constraint propagation, the exploration of the search space must be carefully designed, taking into account specific properties of the considered problem (*e.g.*, dominance relations, symmetries, possible use of decomposition rules).

In conclusion, Chapter 15 mentions various extensions of the model and presents promising research directions.

[2] Most of the material presented in this part comes from *Constraint-Based Scheduling* (Kluwer, 2001) by Ph. Baptiste, C. Le Pape and W. Nuijten

Contents

Part I

Scheduling Equal Length Jobs

Chapter 1

Introduction

Looking through the listing of complexity results for scheduling problems [47] one can observe that the vast majority of problems whose complexity status is unknown involve equal–processing–time jobs. Hence, the state–of–the–art in studying the complexity of scheduling problems has reached the level at which approaches and techniques successfully working for equal–processing–time scheduling become especially important. The first part of this manuscript is dedicated to such problems. We have tried to present, within a common framework, our recent results in this area.

In this introduction we briefly recall (Section 1.1) the Classification Scheme of Graham, Lawler, Lenstra and Rinnooy Kan [113]. We then focus on scheduling problems with identical processing times and we recall some basic results on the special case where processing times equal 1 (Section 1.2). Since many algorithms presented in the following chapters rely on Dynamic Programming, we introduce this technique in Section 1.3. Finally, we provide in Section 1.4 a list of the problems covered in this thesis and we also briefly mention some other results that we have obtained but that are not described in this manuscript.

1.1 Classification Scheme

We use the classification scheme of Graham, Lawler, Lenstra and Rinnooy Kan [113, 38]. It is widely used in scheduling theory and has been extended several times. It allows to represent a large number of scheduling problems under the notation $\alpha|\beta|\gamma$. These parameters describe respectively the machine environment, the job characteristics, and the optimality criterion. Here, we only present the subset of the classification used in this manuscript.

Machine Environment (α) α consists of 2 parameters α_1 and α_2.

- $\alpha_1 = 1$ when a single machine is considered.
- $\alpha_1 = P$ when the machines in the problem are identical.
- $\alpha_1 = Q$ when the machines are uniform, *i.e.*, each machine has a speed that linearly affects the processing times.
- $\alpha_1 = R$ when the machines are unrelated, *i.e.*, the processing times on each machine have to be specified.
- More complex machine environments like Flow-Shop ($\alpha_1 = F$), Job-Shop ($\alpha_1 = J$) or Open-Shop ($\alpha_1 = O$) are also considered.

In the shop problems mentioned, the concept of a *job* is used. The term "job" is used when some operations can be rearranged into a strongly related subset. Typically a job is a set of operations linked by a set of precedence constraints. In the shop problems mentioned, each job J_i consists of a set of operations $\{O_{i,1}, \ldots, O_{i,n_i}\}$, and the machine $\mu_{i,j}$ on which each operation $O_{i,j}$ is to be executed is known in advance. In the Job-Shop Problem, we have chain-like precedence constraints between operations of the same job, *i.e.*, $O_{i,1} \rightarrow O_{i,2} \rightarrow \ldots \rightarrow O_{i,n_i}$. The Flow-Shop Problem is a special case of the Job-Shop Problem where each job has the same machine ordering for its operations, *i.e.*, $\forall i, j\ \mu_{i,j} = j$. In the Open-Shop Problem, there are no precedence constraints but operations of the same job cannot be processed simultaneously.

The second parameter α_2 denotes the number of machines in the problem. If this parameter is omitted, the number of machines is arbitrary, *i.e.*, it is provided in the data of each instance of the problem. For example, $\alpha = P5$ means that there are 5 identical machines while $\alpha = R$ means that there are an unspecified number of unrelated machines.

Job Characteristics (β) As said, the term job is used when some operations can be rearranged into a strongly related subset. Jobs can themselves be related by precedence constraints. The nature of the precedence graph induced by these constraints, if any, is indicated by one of the strings "*chain*", "*tree*", "*prec*", meaning that the precedence graph is a chain, a tree, or any graph.

When jobs are subjected to release dates before which they cannot start, the string "r_i" is added to β. When jobs are subjected to deadlines before which they must be completed, the string "$\bar{d}_i$" is added. "$p_i = p$" and "$p_i = 1$" denote that jobs have identical and unitary processing times, respectively. When preemption is allowed, "*pmtn*" is added to β.

In most of the classical scheduling models, it is assumed that each operation is processed on one machine at a time. However, the relatively recent development of multiprocessor computer systems and of complex manufacturing environments has led researchers to study more complex situations where each operation requires simultaneously several machines (see for instance [35, 88]). We follow the multiprocessor scheduling terminology where "operations" are called "tasks" and "machines" are called "processors".

- The notation $size_i$ is used when for each operation i, a fixed number $size_i$ of processors is required to execute the task (yet, the processors required are not specified).
- The notation fix_i is used when, for each operation i, the set of processors to be used is known in advance.

We also study some batching problems and we follow Brucker and Knust terminology [47]. In a batching problem, jobs start and are completed simultaneously. On a serial batching machine ("*s-batch*"), the length of a batch equals the sum of the processing times of its jobs. On a parallel batching machine ("*p-batch*"), there are at most b jobs per batch and the length of a batch is the largest processing time of its jobs. These environments are detailed in Chapter 5.

Optimality Criterion (γ) In the following, C_i denotes the completion time of the job J_i. Most of the classical scheduling criteria take into account a due date d_i that one would like to meet for each job. In contrast to a deadline $\bar{d}_i$ which is mandatory, a due date d_i can be seen as a preference. The tardiness T_i of J_i is defined as $T_i = \max(0, C_i - d_i)$. The notation U_i is used to denote a unit penalty per late job, *i.e.*, U_i equals 0 when $C_i \leq d_i$ and equals 1 otherwise.

Scheduling criteria F are either formulated as a sum or as a maximum. A weight per job w_i may be used to give more importance to some jobs. We mention the following well-known optimality criteria:

- Makespan: $F = C_{\max} = \max C_i$
- Total weighted flow time: $F = \sum w_i C_i$
- Maximum tardiness: $F = T_{\max} = \max T_i$
- Total weighted tardiness: $F = \sum w_i T_i$
- Total weighted number of late jobs: $F = \sum w_i U_i$

See also Figure 1.1 for some scheduling related objective functions.

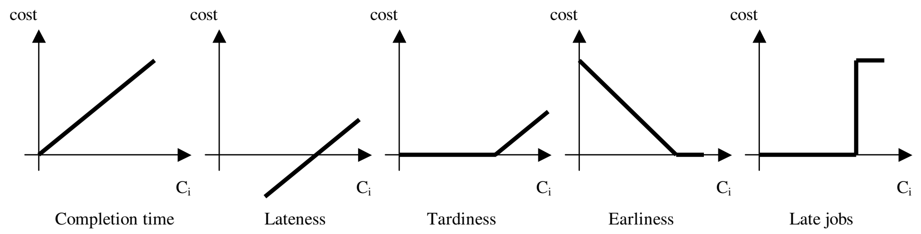


Figure 1.1: Objective functions

1.2 Unit *vs.* Identical Processing Times

When there there is no-precedence constraints, and when the machines are identical, the problem of scheduling unit-processing time jobs, for any regular function F reduces to a simple flow (or assignment) problem.

In the following we assume that $F = \sum f_i$ is a sum function. Consider an optimal active schedule for $P|r_i, p_i = 1|F$. A schedule is said to be active if no job can start earlier without delaying another one or violating its release date. It is easy to see that active schedules are dominant for any regular functions. There are only $O(n)$ relevant time points. This can be seen by considering the two smallest release dates r and r' ($r < r'$). Suppose x jobs have r as release date. If $x \leq r' - r$ then we have at most x relevant time points from r until $r + x$ as it is not useful to schedule any of the x jobs after $r + x$. We can then continue our reasoning with r' and the smallest release date larger than r' (r''). If $x > r' - r$ then we have at most $r' - r$ relevant time points between r and r', and we can continue our reasoning with r' and r'' and by temporarily assigning a release date r' to $x - (r' - r)$ jobs that have an original release date r. By recursively following this reasoning we can see that we will not have more relevant time points than jobs. The cost of scheduling a job J_i at a time point t

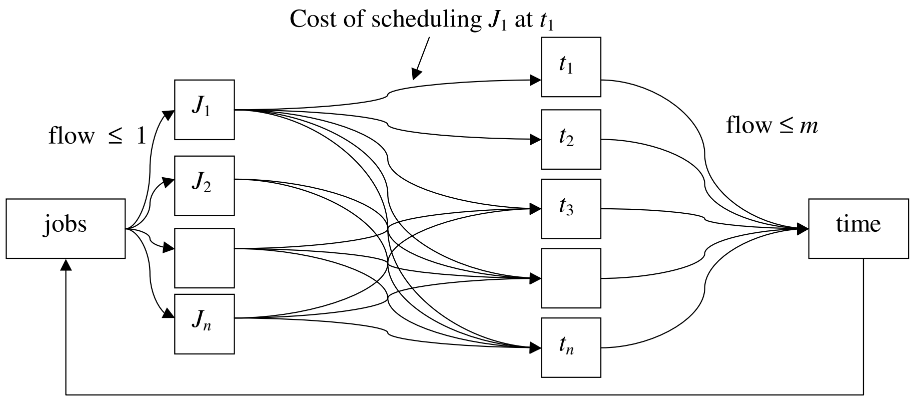


Figure 1.2: Min cost flow for unit execution time schedules.

($r_i \leq t$) is $f_i(t)$. Moreover, at each time point t, less than m jobs can execute. Given the network of Figure 1.2, a maximal flow whose cost is minimal corresponds to an optimal schedule. It can be computed in polynomial time. We refer to [86] for a detailed study of this problem and extensions to uniform parallel machines.

When processing times are equal, the problem cannot, a priori, be reduced to a flow problem. However, if there are no release dates, then on active schedules, start times equal 0 modulo p and so, the problem $P|p_i = p|\sum f_i$ can be reduced to $P|p_i = 1|\sum f'_i$ where $f'_i(t) = f'_i(pt)$. Hence, the flow-based approach also applies.

Although the network flow model does not work in general, many problems with identical processing times are still solvable in polynomial time. We have introduced a set of dynamic programming algorithms for this special case.

1.3 Dynamic Programming

Dynamic Programming [32] is "an algorithmic technique in which an optimization problem is solved by caching subproblem solutions (memoization) rather than recomputing them" [172]. There are two common characteristics to problems that can be solved by dynamic programming.

- An optimal solution can be decomposed into a set of optimal solutions of sub-problems.
- There are "few" relevant sub-problems to consider; where few means polynomially bounded.

The art of Dynamic Programming lies in the definition of the sub-problems to consider. This can be very easy like in the knapsack problem (given items of different values and volumes, find the most valuable set of items that fit in a knapsack of fixed volume) where each sub-problem is defined by two integer values, the first one identifies the subset of items to consider and the second one is the remaining volume of the knapsack. Unfortunately, sub-problems are not always easy to identify and in most of the cases described in the following chapters, we need some more or less complex dominance rules on optimal schedules to identify them.

1.4 Covered Problems

In the following sections, we present a subset of the complexity results that we have recently obtained. Although we study several scheduling environment (single machine, parallel machine, open-shop, *etc.*), almost all the problems discussed in the following chapters have two common characteristics:

- Jobs have identical processing times,
- machines are identical,
- and there are no precedence constraints.

We have also studied several other theoretical scheduling problems ($1|prec, pmtn, p_i = p, r_i|\sum C_i$, $Pm|p_i = 1, intree|\sum C_i$, $Q|p_i = 1, pmtn|\sum U_i$, $P|p_i = 1, chains, r_i|L_{\max}$ [27]; $P2|pmtn, outtree, r_j, p_j = 1|\sum C_j$ [26] and $1|s\text{-}batch|\sum T_i$ [18]) that do not have this three major characteristics. The results obtained on these problems are not reported in this manuscript.

For several of the problems studied in the following sections, we present a solution procedure based on dynamic programming. Such algorithms rely on dominance properties that identify a special job, which, once it is scheduled, allows us to split the problem into two independant sub-problems. The existence of this key job comes from the fact that we can extensively exchange jobs (the absence of precedence consraints and the equal length of jobs play a key role at this point).

Another key observation we use to obtain polynomial–time solutions to non-preemptive problems with equal execution times is that there are very few possible starting times. Indeed, since an optimal non-preemptive schedule is active, *i.e.*, no operation can be shifted to the left without the violation of a constraint, we will see that starting times equal release dates modulo the processing time. In the following, although its definition varies with the exact scheduling environment, Θ always denotes the set of all possible starting and completion times.

Chapter 2

Scheduling Equal Length Jobs on a Single Machine

In this Chapter, we restrict our study to the single machine case $(1|r_i, p_i = p|F)$. We first show (Section 2.1) that when preemption is allowed and when processing times are equal, we can minimize the weighted number of late jobs in polynomial time. The corresponding algorithm has been published in [12]. We have a slightly more general result (Section 2.2) for the non-preemptive case since our dynamic program can handle any ordered objective function (see Definition 2). Hence we show that $\sum w_i U_i$, $\sum w_i C_i$ and $\sum T_i$ can be minimized in polynomial time. These results have been published in [13].

2.1 Preemptive Problem, $1|r_i, pmtn, p_i = p| \sum w_i U_i$

$1|pmtn, r_j, \sum w_j U_j$ is NP-hard but can be solved in pseudo-polynomial time by Lawler's algorithm [142] whose time and space complexities are respectively $O(nk^2W^2)$ and $O(k^2W)$, where k is the number of distinct release dates and where W is the sum of the weights of the jobs. If weights are equal $1|pmtn, r_j| \sum U_j$, the problem obviously becomes strongly polynomial; the time and space bounds of Lawler's algorithm reducing to $O(n^3k^2)$ and $O(nk^2)$. So, $O(n^5)$ and $O(n^3)$ if all release dates are distinct. In [14], we describe an algorithm for this special case that improves the bounds to respectively $O(n^4)$ and $O(n^2)$.

When processing times are equal, the status of the weighted problem was still open. In the following, we describe a dynamic programming from [12] that solves this special case.

2.1.1 Problem Reformulation

Let O be a subset of jobs and let JPS be the preemptive schedule of these jobs associated to the EDD dispatching rule: whenever the machine is free and one job is available, schedule the job J_i for which $\bar{d}_i$ is the smallest. If a job J_j becomes available while J_i is in process, stop J_i and start J_j if $\bar{d}_j$ is strictly smaller than $\bar{d}_i$; otherwise continue J_i. Jackson Preemptive Schedule has several interesting properties (*e.g.*, [55]). In particular, if a job is scheduled on JPS after its due date, there is no preemptive schedule of O where all jobs are on-time. In the following, O is said to be feasible if and only if all jobs are on-time in its JPS. Given this definition, searching for a schedule on which the weighted number of late jobs is minimal, reduces to finding a set of jobs whose weight is maximal and whose JPS is feasible.

2.1.2 Dominance Property

Proposition 1. *For any subset of jobs Z, the start and end times of the jobs on the JPS of Z belong to the set Θ.*

$$\Theta = \{t | \exists r_i, \exists l \in \{0, \dots, n\}, t = r_i + lp\}.$$

Proof. We first prove that the end time of a job on the JPS of Z belongs to Θ. Let J_k be any job and let s and e be respectively its start and end times on JPS. Let t be the minimal time point such that between t and s JPS is never idle. Because of the structure of JPS, t is a release date, say r_x. The jobs that execute (even partially) between s and e do not execute before s nor after e (because Jackson Preemptive schedule is based upon the EDD rule). Thus $e - s$ is a multiple of p. Two cases can occur:

- Either J_k causes an interruption and hence $s = r_k$.
- Or J_k does not cause any interruption and hence the jobs that execute between r_x and s, are fully scheduled in this interval. Consequently, $s - t$ is a multiple of p.

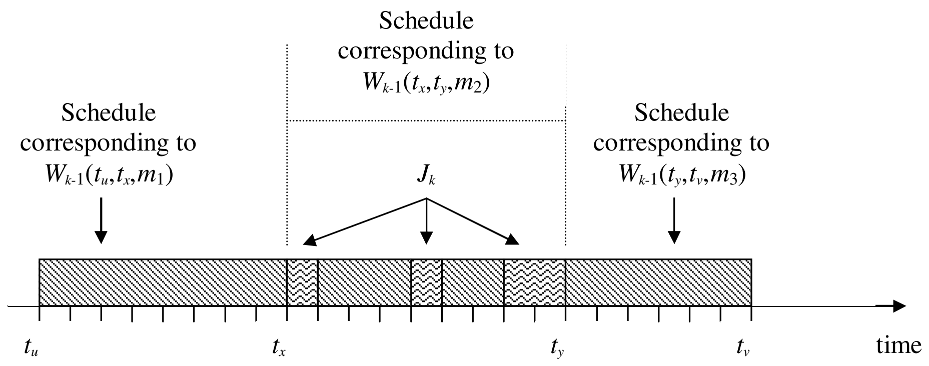
In both cases, there is a release date r_y (either r_k or r_x) such that between r_y and e, JPS is never idle and such that e is equal to r_y modulo p. On top of that, the distance between r_y and e is not greater than np (because JPS is not idle). Hence, $e \in \Theta$.

Now consider the start time of any job on JPS. This time point is either the release date of the job or is equal to the end time of the "previous" one. Thus, start times also belong to Θ. □

Definition 1. *For any time points t_u, t_v in Θ with $u < v$ and for any integer value k such that $1 \leq k \leq n$,*

- *let $U_k(t_u, t_v) = \{J_i | i \leq k, t_u \leq r_i < t_v\}$*
- *and for any $m \leq n$, let $W_k(t_u, t_v, m)$ be the maximal weight of a subset $Z \subseteq U_k(t_u, t_v)$ of m jobs such that, the JPS of Z is feasible and ends before t_v. If there is no such subset, $Wk(t_u, t_v, m)$ is set to $-\infty$.*

In the following, we note $\Theta = \{t_1, t_2, ..., t_q\}$ the ordered set of distinct time-points in Θ. Recall that $q \leq n^2$.



Schedule corresponding to $W_{k-1}(t_x,t_y,m_2)$
Schedule corresponding to $W_{k-1}(t_u,t_x,m_1)$
J_k
Schedule corresponding to $W_{k-1}(t_y,t_v,m_3)$
t_u
t_x
t_y
t_v
time

Figure 2.1: Preemptive Schedule of J_k

Proposition 2. *For any time points $t_u, t_v \in \Theta$ with $u < v$ and any integer values k and m such that $1 < k \leq n$ and $1 \leq m \leq n$, $W_k(t_u, t_v, m)$ can be computed as follows:*
If $r_k \notin [t_u, t_v)$, $W_k(t_u, t_v, m) = W_{k-1}(t_u, t_v, m)$. Otherwise, $W_k(t_u, t_v, m) = \max(W_{k-1}(t_u, t_v, m), W')$ where W' is the maximum of

$$W_{k-1}(t_u, t_x, m_1) + W_{k-1}(t_x, t_y, m_2) + W_{k-1}(t_y, t_v, m_3) + w_k$$

under the constraints
$$\begin{cases} t_x, t_y \in \Theta \\ \max(r_k, t_u) \le t_x < t_y \le \min(d_k, t_v) \\ m_1 + m_2 + m_3 = m - 1 \\ p(m_2 + 1) = t_y - t_x \end{cases}$$

Proof. It is obvious that if $r_k \notin [t_u, t_v)$, $W_k(t_u, t_v, m)$ is equal to $W_{k-1}(t_u, t_v, m)$. In the following, we suppose that $r_k \in [t_u, t_v)$.

We first prove that $\max(W_{k-1}(t_u, t_v, m), W') \le W_k(t_u, t_v, m)$.

- Case 1: $W_{k-1}(t_u, t_v, m) > W'$. Since $U_{k-1}(t_u, t_v) \subseteq U_k(t_u, t_v)$, we have
$$\max(W_{k-1}(t_u, t_v, m), W') \le W_k(t_u, t_v, m).$$

- Case 2: $W' > W_{k-1}(t_u, t_v, m)$. There exist $t_x \in \Theta, t_y \in \Theta$ and 3 integers m_1, m_2, m_3 such that
$$\begin{cases} \max(r_k, t_u) \le t_x < t_y \le \min(d_k, t_v) \\ m_1 + m_2 + m_3 = m - 1 \\ p(m_2 + 1) = t_y - t_x \\ W' = W_{k-1}(t_u, t_x, m_1) + W_{k-1}(t_x, t_y, m_2) + W_{k-1}(t_y, t_v, m_3)) + w_k \end{cases}$$
Obviously, the subsets $U_{k-1}(t_u, t_x), U_{k-1}(t_x, t_y)$ and $U_{k-1}(t_y, t_v)$ do not intersect. Thus, the JPS schedules of the subsets that realize $W_{k-1}(t_u, t_x, m_1)$, $W_{k-1}(t_x, t_y, m_2)$ and $W_{k-1}(t_y, t_v, m_3)$, put one after another define a valid overall schedule of a set of $m-1$ jobs taken in $U_{k-1}(t_u, t_v)$. Moreover, between t_x and t_y there is enough space to schedule J_k since m_2 jobs in $U_{k-1}(t_x, t_y)$ are scheduled and since $p(m_2 + 1) = t_y - t_x$ (see Figure 2.1). As a consequence, $W' \le W_k(t_u, t_v, m)$.

Now, we prove that $W_k(t_u, t_v, m) \le \max(W_{k-1}(t_u, t_v, m), W')$. We only consider the case where $W_k(t_u, t_v, m)$ is finite otherwise the claim holds. Consider a set Z that realizes $W_k(t_u, t_v, m)$. If J_k does not belong to Z then $W_k(t_u, t_v, m) = W_{k-1}(t_u, t_v, m) \le \max(W_{k-1}(t_u, t_v, m), W')$. Suppose now that $J_k \in Z$. Let t_x and t_y be the start and end times of J_k on the JPS of Z. Thanks to Proposition 1, we know that $t_x \in \Theta$ and $t_y \in \Theta$. We also have $\max(r_k, t_u) \le t_x < t_y \le \min(d_k, t_v)$. Let Z_1, Z_2, Z_3 be the partition of $Z - \{J_k\}$ into the jobs that have a release date between t_u and t_x, between t_x and t_y and between t_y and t_v. Because of the structure of JPS(J_k is the job whose due date is maximal), all jobs in Z_1 are completed before t_x. Moreover, all jobs in Z_2 start after t_x and are completed before t_y, and all jobs in Z_3 are completed before t_v. On top of that, $p(|Z_2|+1) = t_y - t_x$ because J_k is also scheduled between t_x and t_y. Moreover, we have $|Z_1| + |Z_2| + |Z_3| + 1 = m$. Finally the weight of Z_1 is not greater than $W_{k-1}(t_u, t_x, |Z_1|)$, the weight of Z_2 is not greater than $W_{k-1}(t_x, t_y, |Z_2|)$ and the weight of Z_3 is not greater than $W_{k-1}(t_y, t_v, |Z_3|)$. This leads to $W_k(t_u, t_v, m) \le \max(W_{k-1}(t_u, t_v, m), W')$. $\square$

Our dynamic programming algorithm relies on the above proposition. The values of $W_k(t_u, t_v, m)$ are stored in a multi-dimensional array of size $O(n^6)$ (n possible values for k, n^2 possible values for t_u, n^2 possible values for t_u, and n possible values for m).

- In the initialization phase the value of $W_1(t_u, t_v, m)$ is set to w_1 if $m = 1$ and if p is not greater than $\min(d1, tv) - \max(r1, tu)$ and to $-\infty$ otherwise.

- We then iterate from $k = 2$ to $k = n$. Each time, W_k is computed for all the possible values of the parameters thanks to the formula of Proposition 2 and to the values of W_{k-1} computed at the previous step.

The maximum weighted number of on-time jobs is equal to
$$\max(W_n(t_0, t_q, 1), W_n(t_0, t_q, 2), \ldots, W_n(t_0, t_q, n)).$$

The overall complexity of the algorithm is $O(n^5)$ for the initialization phase. For each value of k, $O(n^5)$ values of W_k have to be computed. For each of them, a maximum among $O(n^4)$ terms has to be computed (for given values of t_x, m_1 and m_2, there is only one possible value for both t_y and m_3). This leads to an overall time complexity of $O(n^10)$. A rough analysis of the space complexity leads to an $O(n^6)$ bound but since, at each step of the outer loop on k, one only needs the values of W computed at the previous step $(k-1)$, the algorithm can be implemented with 2 arrays of $O(n^5)$ size (one for the current values of W and one for the previous value of W).

2.2 Non-Preemptive Problems, $1|r_i, p_i = p| \sum f_i(C_i)$

In this section, we consider the scheduling situation where n jobs $J_1, \dots, J_n$ have to be scheduled on a single machine. Each job is described by a release date r_i, a processing time p_i and a cost function $f_i(t)$. This function represents the cost induced by J_i when it is completed at time t. The problem of minimizing the sum of the f_i functions consists of finding a set of completion times C_i for each job J_i such that (1) jobs start after their release date, *i.e.*, $\forall i, C_i - p_i \geq r_i$, (2) jobs do not overlap, and (3) the objective function $\sum_i f_i(C_i)$ is minimal. In the following, a schedule meeting (1) and (2) is said to be feasible.

It is shown in [53, 55, 109] that the problem of minimizing the number of late jobs ($f_i = U_i$) is polynomial as soon as processing times are equal, while the general problem is NP-hard [108]. Simons ([201]) describes a polynomial algorithm to minimize the sum of the completion times when processing times are equal. Note that Simon's algorithm with parallel machines too and that in the single machine case, arbitrary precedence constraints can be taken into account.

These results leave open the status of $1|r_i, p_i = p| \sum w_i U_i$ $1|r_i, p_i = p| \sum w_i C_i$, $1|r_i, p_i = p| \sum T_i$ and $1|r_i, p_i = p| \sum w_i T_i$. In the following, we show that single machine problems with identical processing times can be solved polynomially for a large class of so called **ordered objective functions**. All these results are extend to parallel machines problems in Chapter 3.

Definition 2. *F is an ordered objective function if and only if :*

1. *F is a sum function, i.e.,* $F = \sum f_i(C_i)$,
2. *F is regular, i.e.,* $\forall i, f_i$ *is non-decreasing,*
3. f_i *is constant after a time point* $\boldsymbol{\delta_i}$*, i.e.,* $\forall t > \delta_i, f_i(t) = \omega_i$,
4. $\forall i < j$, $\delta_i \leq \delta_j$ *and* $t \mapsto (f_i - f_j)(t)$ *is non-decreasing over* $[0, \delta_i]$.

The last condition of the definition ensures that the function has a "Monge"-like property. For such functions, it is known (*e.g.* [38]) that many unit execution time scheduling problems are polynomially solvable.

It is easy to verify that the weighted number of late jobs, $\sum w_i U_i$, is an ordered objective function. On the contrary, $\sum w_i C_i$ and $\sum T_i$ are not ordered objective functions. However, conditions 1 and 2 of Definition 2 hold for these functions and jobs can be renumbered to meet condition 4.

We show how a function like $\sum w_i C_i$ or $\sum T_i$ can be modified, without changing the optimum value, to become an ordered objective function: Consider a large time point T and alter the functions f_i after T so that $\forall t \geq T, f_i(t) = M$, where M is another large value. If T and M are large enough, the optimum of the modified problem is also the optimum of the original one. Moreover, the modified functions are ordered objective functions.

From now on, we restrict our study to ordered objective functions and we have $\delta_1 \leq \dots \leq \delta_n$. By analogy with due date scheduling, we say that a job is late when it is completed after δ_i and that it is on-time otherwise. The "late" cost is ω_i and the early cost is time-dependant. Notice that a late job can be scheduled arbitrary late.

We first introduce a dominance property (Section 2.2.1) on starting times and we describe the variables of the dynamic programming algorithm and the Theorem that links them together (Section 2.2.2). The dynamic programming algorithm itself is described in Section 2.2.3.

2.2.1 Dominance Property

Proposition 3 provides a characterization of the time points at which jobs start and end on active schedules.

Proposition 3. *In active schedules, start and completion times belong to*

$$\Theta = \{t | \exists r_i, \exists l \in \{0, \ldots, n\}, t = r_i + lp\}.$$

Proof. Consider an active schedule and a job J_k. Let t be the largest time point, before the start time of J_k, at which the machine is idle immediately before t. If t is not a release date, the job scheduled immediately after t could be scheduled earlier and thus the schedule would not be active. t is then a release date, say r_i. Between r_i and the starting time of J_k, l jobs execute ($0 \leq l \leq n-1$). Hence the starting time and the ending time of J_k belong to Θ. □

Since active schedules are dominant for regular objective functions, we will restrict our search to schedules where starting times belong to Θ.

2.2.2 Variables Definition and Decomposition Scheme

Recall that late jobs can be scheduled arbitrary late. Hence, there is no point to assign a starting time to late jobs. In the following, when we refer to a schedule $\mathcal{H}$ of some subset of jobs O, it does not mean that all jobs in O are in the schedule (the on-time jobs only are in the schedule). So the "cost" of $\mathcal{H}$ with respect to O is

- the sum, over all jobs $J_i \in O$ scheduled in $\mathcal{H}$, of $f_i(C_i)$
- plus the sum, over all other jobs $J_i \in O$ not scheduled in $\mathcal{H}$, of ω_i.

For any integer $k \leq n$, let $U_k(s, e)$ be the set of jobs whose index is lower than or equal to k and whose release date is in the interval $[s, e)$ (same definition as in the preemptive case). Let $F_k(s, e)$ be the minimal cost over all feasible schedules $\mathcal{H}$ of the jobs in $U_k(s - p, e)$ such that

1. no machine is used before s on $\mathcal{H}$,
2. no machine is used after e on $\mathcal{H}$,
3. starting times of jobs on $\mathcal{H}$ belong to Θ.

$F_k(s, e)$ is always defined since an empty schedule (*i.e.*, a schedule where all jobs of $U_k(s - p, e)$ are late) is a feasible schedule and meets Constraints 1, 2 and 3. Notice that given our definition, $F_0(s, e)$ is equal to 0.

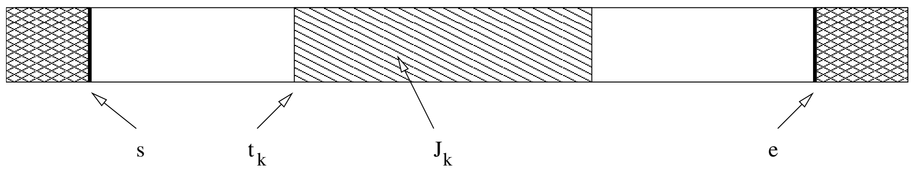


Figure 2.2: Relative positions of J_k (starting at t_k), s and e.

Proposition 4. *Let $k \in [1, n]$ and let $[s, e)$ be any time interval. If $r_k \notin [s - p, e)$, $F_k(s, e) = F_{k-1}(s, e)$. If $r_k \in [s - p, e)$, then $F_k(s, e)$ is equal to $\min(\omega_k + F_{k-1}(s, e), F')$; where*

$$F' = \min_{\substack{\max(s, r_k) \leq t_k \\ t_k + p \leq \min(e, \delta_k) \\ t_k \in \Theta}} F_{k-1}(s, t_k) + F_{k-1}(t_k + p, e) + f_k(t_k + p).$$

(If F' is undefined, assume that $F' = \infty$.)

Proof. If $r_k \notin [s-p, e)$ the proposition holds because $U_k(s-p,e) = U_{k-1}(s-p,e)$. Now, assume that $r_k \in [s-p,e)$.
We first prove that $\omega_k + F_{k-1}(s,e) \geq F_k(s,e)$ and that $F' \geq F_k(s,e)$.

- The schedule that realizes $F_{k-1}(s,e)$ and where J_k is late is feasible for the set of jobs $U_k(s-p,e)$. The cost of this schedule is $\omega_k + F_{k-1}(s,e)$. So, we have $\omega_k + F_{k-1}(s,e) \geq F_k(s,e)$

- Assume that F' is finite. According to the definition of F', there exists a time point t_k such that (1) $t_k \geq \max(s, r_k)$, (2) $t_k \leq \min(e, \delta_k) - p$ and (3) $F' = F_{k-1}(s,t_k) + F_{k-1}(t_k,e) + f_k(t_k+p)$. Let $\mathcal{H}_1$ and $\mathcal{H}_2$ be two schedules that realizes respectively $F_{k-1}(s,t_k)$ and $F_{k-1}(t_k+p,e)$. Notice that any job in $U_{k-1}(s-p,e)$ is either late or scheduled in $\mathcal{H}_1$ or in $\mathcal{H}_2$. Consider the schedule $\mathcal{H}$ built as follows: schedule J_k at time t_k and "add" $\mathcal{H}_1$ and $\mathcal{H}_2$. Given the definition of F_{k-1}, $\mathcal{H}$ is a feasible schedule of $U_k(s-p,e)$, *i.e.*, jobs do not overlap in time and start (after their release dates) at time points that obviously belong to Θ. On top of that, $\mathcal{H}$ is idle before s and after e. Since $t_k + p \leq \delta_k$, J_k is on-time and thus, the cost of $\mathcal{H}$ is exactly F'. Hence, $F' \geq F_k(s,e)$.

We now prove that $\min(\omega_k + F_{k-1}(s,e), F') \leq F_k(s,e)$. Consider a schedule that realizes $F_k(s,e)$ and let O be the set of jobs scheduled in this schedule (jobs in $O - U_k(s-p,e)$ are late). Among all schedules (1) that realize $F_k(s,e)$ and (2) in which the same set of jobs O are scheduled, consider the schedule $\mathcal{H}$ that lexicographically minimizes the vector made of completion times of jobs in O. (The completion time of the job in O, with the smallest index, is minimum, then the completion time of the job in O, with the second smallest index, *etc.*). The job J_k is either late or on-time on $\mathcal{H}$.

- If J_k is late then $\mathcal{H}$ is also a feasible schedule for $U_{k-1}(s-p,e)$ and its cost is exactly $F_k(s,e) - \omega_k$ (the late cost ω_k is removed since J_k is not considered any longer). Thus, $F_k(s,e) - \omega_k \geq F_{k-1}(s,e)$ and our claim holds.

- Now assume that J_k is on-time and let $t_k \in \Theta$ be the starting time of J_k in $\mathcal{H}$. Note that $t_k + p \leq \min(e, \delta_k)$ and $\max(s, r_k) \leq t_k$. In the following, we show that this starting time allows us to "partition" all jobs.
Suppose that there is a job J_i with $r_i \leq t_k$ that is executed on-time after J_k, at time t_i in $\mathcal{H}$ ($t_k \leq t_i$). Let $\mathcal{H}'$ be the schedule obtained from $\mathcal{H}$ by exchanging the jobs J_i and J_k. The exchange is valid because $r_i \leq t_k$ and moreover, both jobs remain on-time because $\delta_i \leq \delta_k$. The relative cost Δ of the exchange is
$$\begin{aligned}\Delta &= f_i(t_k+p) + f_k(t_i+p) - (f_i(t_i+p) + f_k(t_k+p)) \\ &= (f_i - f_k)(t_k+p) - (f_i - f_k)(t_i+p).\end{aligned}$$
Note that $t_i + p \leq \delta_i$ because J_i is on-time. Moreover, $i < k$ and thus $t \mapsto (f_i - f_k)(t)$ is non-decreasing over $[0, \delta_i]$. Hence $\Delta \leq 0$. Given the definition of $F_k(s,e)$, Δ cannot be strictly negative (this would contradict the fact that $\mathcal{H}$ realizes the minimum). Hence $\mathcal{H}$ and $\mathcal{H}'$ have the same cost. However, $\mathcal{H}'$ is better than $\mathcal{H}$ for the lexicographical order. This contradicts our hypothesis on $\mathcal{H}$. *Hence, all jobs with a release date lower than or equal to t_k are late or are scheduled before t_k.*
Now, let us decompose $\mathcal{H}$. Let $\mathcal{H}_1$ and $\mathcal{H}_2$ denote the left and the right parts of $\mathcal{H}$ (before t_k and after $t_k + p$). The cost of $\mathcal{H}$ is the sum of the costs of $\mathcal{H}_1$ and $\mathcal{H}_2$ plus $f_k(t_k+p)$. Since all jobs with a release date lower than or equal to t_k are late or are scheduled before t_k, the cost of $\mathcal{H}_1$ is greater than $F_{k-1}(s,t_k)$ and the cost of $\mathcal{H}_2$ is greater than $F_{k-1}(t_k+p,e)$. Hence, $F_k(s,e) \geq F_{k-1}(s,t_k) + F_{k-1}(t_k+p,e) + f_k(t_k+p)$.

□

2.2.3 A Dynamic Programming Algorithm

Active schedules are dominant so, the optimum is exactly $F_n(\min_{t\in\Theta} t, \max_{t\in\Theta} t)$. Thanks to Theorem 4, we have a straight dynamic programming algorithm to reach the optimum. The relevant values for s and e are exactly those in Θ. The values of $F_k(s,e)$ are stored in a multi-dimensional array of size $O(n^5)$ (n possible values for k, n^2 possible values both for s and e). Our algorithm works as follows:

- In the initialization phase, $F_0(s,e)$ is set to 0 for any values s, e in Θ ($s \leq e$).
- We then iterate from $k = 1$ to $k = n$. Each time, F_k is computed for all the possible values of the parameters thanks to the formula of Theorem 4, and to the values of F_{k-1} computed at the previous step.

The initialization phase runs in $O(n^4)$ because the size of Θ is upper bounded by n^2. Afterwards, for each value of k, $O(n^4)$ values of $F_k(s,e)$ have to be computed. For each of them, a maximum among $O(n^2)$ terms is computed (because there are $O(n^2)$ possible values for $t_k \in \Theta$). This leads to an overall time complexity of $O(n^7)$. A rough analysis of the space complexity leads to an $O(n^5)$ bound but since, at each step of the outer loop on k, one only needs the values of F computed at the previous step ($k-1$), the algorithm can be implemented with 2 arrays of $O(n^4)$ size: one for the current values of F and one for the previous values of F. (To build the optimal schedule, all values of $F_k(s,e)$ have to be kept; hence the initial $O(n^5)$ bound applies.)

Chapter 3

Scheduling Equal Length Jobs on Parallel Machines

In this Chapter, we consider the situation where m identical parallel machines are available. In the **preemptive case** (Section 3.1), we show that, if all jobs are released at time 0, the total tardiness problem can be solved in polynomial time thanks to a simple LP formulation of the problem. We also present a dynamic program to minimize the weighted number of late jobs. In the **non-preemptive case** (Section 3.2), we generalize the results obtained on a single machine and we show that $\sum w_i U_i, \sum w_i C_i$ and $\sum T_i$ can be minimized in polynomial time for any fixed value of m. These results have been published in [13]. Finally, we study the weighted number of late jobs criteria in a **Unit Execution Time open-shop** environment $Om|p_{ij} = 1, r_i| \sum w_i U_i$ (Section 3.3). We use a result from Brucker, Jurisch and Jurisch to transform the UET open-shop into a preemptive parallel machine problem and we show that it can be solved in polynomial time with a rather complex Dynamic Program.

3.1 Preemptive Problems

3.1.1 Maximal Lateness, $P|pmtn|L_{\max}$

We study the scheduling situation where n jobs $J_1, \cdots, J_n$ with processing times $p_1, \cdots, p_n$ have to be preemptively scheduled on m parallel identical machines. Preemption means that jobs can be interrupted at any time and resumed later on either machine. A Deadline $\bar{d}_i$ is associated to each job J_i and the question is whether there is a schedule meeting all constraints or not ($P|pmtn, \bar{d}_i| - -$).

Deadline Scheduling. Horn [122] has shown that there is a feasible schedule if and only if $\forall i, p_i \leq \bar{d}_i$ and $\forall i, \sum_j \max(0, p_j - \max(0, \bar{d}_j - \bar{d}_i)) \leq m\bar{d}_i$. Sahni's algorithm builds a schedule [197] in $O(n \log mn)$ steps. Sahni and Cho [198] have shown that the same problem on uniform machines (where each machine execute jobs at its own speed) could be solved in $O(n \log n + mn)$.

Due-Date Scheduling. The $L_{\max}$ problem is known to be solvable in $O(n \log m + mn)$ thanks to an Algorithm of Labetoulle, Lawler, Lenstra and Rinnoy Kan [136]. Actually this algorithm also works for $Q|pmtn|L_{\max}$ the problem where machines are uniform.

In the following, we describe an $O(n \log n)$ algorithm for $P|pmtn|L_{\max}$. This improves on the previous ones for large values of m. To build the optimal schedule, our algorithm works as follows:

1. The optimum $L_{\max}$ value is computed,
2. deadlines are set to the due-dates value plus $L_{\max}$ and the corresponding schedule is built thanks to Sahni's algorithm

The crucial step is the computation of $L_{\max}$ (Sahni's algorithm runs in $O(n \log mn)$ steps so, step 2 runs in $O(n \log n)$).

When the maximal lateness $L_{\max}$ is fixed, one can simply increase due-dates of $L_{\max}$ and use the Horn's [122] conditions to test whether there is a solution with such a maximal lateness. Hence, to reach the minimum maximal lateness, one has to compute the minimal value $L_{\max}$ such that:

$$\forall i, p_i \leq d_i + L_{\max} \tag{3.1}$$

$$\forall i, \sum_j \max(0, p_j - \max(0, d_j - d_i)) \leq m(d_i + L_{\max}) \tag{3.2}$$

Actually, this minimum value $L_{\max}$ is exactly equal to

$$\max\left(\max_i p_i - d_i, \max_i \left\lceil \frac{\sum_j \max(0, p_j - \max(0, d_j - d_i))}{m} \right\rceil - d_i\right) \tag{3.3}$$

Algorithm 1 computes the values $\sum_j \max(0, p_j - \max(0, \bar{d}_j - \bar{d}_i))$ for all i in $O(n \log n)$. We introduce the function $W(t)$

$$W(t) = \sum_j \max(0, p_j - \max(0, \bar{d}_j - t)) \tag{3.4}$$

Note that the function $W(t)$ is piecewise linear, continuous and non-decreasing. Moreover, the time points at which the slope changes are either deadlines or $\bar{d}_i - p_i$ values. We first build two ordered lists $\mathcal{D}$ and $\mathcal{DP}$ of integer values that contain respectively the deadlines and the $\bar{d}_i - p_i$ values. Two indices $i_{\mathcal{D}}$ and $i_{\mathcal{DP}}$ are used to iterate over these lists. $\mathcal{D}$ and $\mathcal{DP}$ are merged in a list $\mathcal{T}$ that contains all relevant time points. The algorithm starts at time 0 and iterate over relevant time points in $\mathcal{T}$. Between two of them, the slope of the function $W(t)$ does not change and when we encounter a deadline, it is easy to see that the slope decreases of 1 while when a $\bar{d}_i - p_i$ value is encountered and that the slope increases of 1 when a deadline is met (*cf.*, the "while" loops of Algorithm 1 lines 11–14 and 15–18). Once the new slope is computed one has simply to increase W of *slope*$(t - t^{\text{old}})$, t^{old} being the previous value taken by t.

Algorithm 1 Computation of $W(t)$ for $t \in \mathcal{T}$

```
1: $\mathcal{D} := \emptyset, \mathcal{DP} := \emptyset$
2: for $i = 1$ to $n$ do
3:     Add $\bar{d}_i$ to $\mathcal{D}$, add $\bar{d}_i - p_i$ to $\mathcal{DP}$
4: end for
5: Sort $\mathcal{D}$ and $\mathcal{DP}$ in non-decreasing order
6: Merge $\mathcal{D}$ and $\mathcal{DP}$ into $\mathcal{T}$ and remove duplicated values
7: $W := 0, t^{\text{old}} := 0, slope := 0$
8: $i_{\mathcal{D}} := 1, i_{\mathcal{DP}} := 1$
9: for $t \in \mathcal{T}$ do
10:    $W := W + slope(t - t^{\text{old}})$          at this point, $W(t)$ equals $W$
11:    while $\mathcal{DP}[i_{\mathcal{DP}}] = t$ do
12:        $slope := slope + 1$
13:        $i_{\mathcal{DP}} := i_{\mathcal{DP}} + 1$
14:    end while
15:    while $\mathcal{D}[i_{\mathcal{D}}] = t$ do
16:        $slope := slope - 1$
17:        $i_{\mathcal{D}} := i_{\mathcal{D}} + 1$
18:    end while
19:    $t^{\text{old}} := t$
20: end for
```

The time complexity of Algorithm 1 is $O(n \log n)$ since (1) the initial sorting requires $O(n \log n)$ steps, (2) merging two lists can be done in linear time and (3) the total number of iterations performed inside the "while" loops equals the size of the lists, *i.e.*, $O(n)$. This algorithm is closely related to the algorithm designed in Section 9.2.3 for computing the "Required Energy Consumption" of a set of jobs over time.

3.1.2 Total Tardiness, $P|pmtn, p_i = p|\sum T_i$

Brucker, Heitmann and Hurink [40] have shown that preemption is redundant for $P|pmtn, p_j = 1|\sum T_j$. Hence the problem can be reduced to a simple assignment problem. We introduce a more general technique applicable to a wider range of problems. Let us consider the class of problems $P|pmtn, p_j = p|\sum f_j$, where f_j are convex nondecreasing functions such that differences $f_i - f_j$ are all monotone functions. Since tardiness $T_j = \max\{0, C_j - d_j\}$ meet these conditions, $P|pmtn, p_j = p|\sum T_j$ is in the class. Note that $\sum T_j$ is a piecewise linear function. We assume that the jobs are in a linear order where for each pair of jobs J_i and J_j the functions $f_i - f_j$ are either strictly increasing or constant if $i < j$. As it is shown in [15], such an order always exists.

It can be trivially shown by the exchange argument that for any problem in the class there exists an optimal schedule where $i \le j \Rightarrow C_i \le C_j$. Let us consider completion times C_j for all $j = 1, 2, \ldots, n$ as deadlines. It is known [122] that a feasible schedule for the decision problem $P|pmtn$ with deadlines C_j exits if and only if (see Equation 3.3)

$$\forall j : \textstyle\sum_{i=1}^{n} \max\{0, p_i - \max\{0, C_i - C_j\}\} \le C_j m \quad \text{and} \quad C_j \ge p_j.$$

Under the conditions $p_j = p$ and $i \le j \Rightarrow C_i \le C_j$ this predicate is

$$\forall j : \textstyle\sum_{i=j+1}^{n} \max\{0, p - C_i + C_j\} \le mC_j - jp \quad \text{and} \quad C_j \ge p.$$

Introducing additional variables $X_{ij} = \max\{0, p - C_i + C_j\}$ for all $i = 2, 3 \ldots, n$ and $j = 1, 2, \ldots, n$ we finally obtain the convex program of minimizing $\sum_{j=1}^{n} f_j(C_j)$ under the linear constraints

$$\begin{array}{rrl} \forall j : & \sum_{i=j+1}^{n} X_{ij} \le mC_j - jp & \text{and} \quad C_j \ge p, \\ \forall i : \forall j : & X_{ij} \ge p - C_i + C_j & \text{and} \quad X_{ij} \ge 0. \end{array}$$

For any convex piecewise linear objective function it can be solved in polynomial time. Once the optimal completion times are known, the optimal schedule can be produced by Sahni's algorithm [197].

3.1.3 Weighted Number of Late Jobs, $Pm|pmtn, p_i = p|\sum w_i U_i$

In Section 3.1.3 we propose a variant of Sahni's Algorithm for $Pm|pmtn, \bar{d}_i|-$. The particular structure of the algorithm allows us to derive a pseudopolynomial Algorithm for $Pm|pmtn|\sum w_i U_i$ (Section 3.1.3). We will see (Section 3.1.3) that when processing times are equal, the Algorithm runs in polynomial time.

An Algorithm to Build a Feasible Schedule

Given a set of deadlines, we describe an algorithm that builds a feasible schedule (if one exists). Informally speaking, the algorithm iterates over jobs and at each step, the current job is scheduled. The first machine is loaded as much as possible, then the second, *etc.* It is **very similar** to the algorithm of **Sahni** [197] but the structure of the schedule is slightly different and will be of prime interest for further developments in this chapter.

For each machine u and each job J_i, let p_i^u be the number of time units of J_i scheduled on machine u and let C_i^u denote the completion time of the piece of J_i scheduled on machine u. By convention, C_i^u is set to C_{i-1}^u if J_i is not being processed on machine u. Throughout one iteration, p_i^u and C_i^u are computed and the processing time as well as the deadline of the job are decreased by p_i^u (*cf.*, Algorithm 2).

It is easy to understand that if the algorithm terminates, all jobs are fully scheduled and the schedule is feasible (decreasing the deadline of J_i at each step ensures that there is no overlap in time of the same job). Notice that structure of the algorithm implies that $C_i^1 \ge C_i^2 \ge \cdots \ge C_i^m$ and that machines are never idle. Such schedules are said to have a "staircase" structure.

Algorithm 2 Scheduling on-time $\{J_1, \cdots, J_n\}$

```
1:  for $u = 1$ to $m$ do
2:      $C_0^u := 0$        initialization with a fake job
3:  end for
4:  for $i = 1$ to $n$ do
5:      for $u = 1$ to $m$ do
6:          $p_i^u := \max(0, \min(p_i, \bar{d}_i - C_{i-1}^u))$
7:          Schedule $p_i^u$ units of $J_i$ between $C_{i-1}^u$ and $C_{i-1}^u + p_i^u$ on machine $u$
8:          $p_i := p_i - p_i^u$
9:          $\bar{d}_i := \bar{d}_i - p_i^u$
10:         $C_i^u := C_{i-1}^u + p_i^u$
11:     end for
12:     if $p_i > 0$ then
13:         All jobs cannot be on-time, exit
14:     end if
15: end for
```

Now, we prove that if the algorithm does not terminate, there is no feasible schedule. In the following, $\mathcal{S}_i$ denotes the schedule built by the algorithm at iteration i and for any schedule $\mathcal{S}$, $\mathcal{S}(t)$ denotes the set of jobs that are being processed at t on $\mathcal{S}$. We say that a schedule $\mathcal{Q}$ of a subset of jobs can be extended if and only if there is a feasible schedule $\mathcal{S}$ of $J_1, \cdots, J_n$ on which all jobs of $\mathcal{Q}$ are scheduled at the same time-points as in $\mathcal{S}$.

Proposition 5. *If $\mathcal{S}_{i-1}$ can be extended to a feasible schedule, $\mathcal{S}_i$ can also be extended to a feasible schedule.*

Proof. Let $\mathcal{S}$ be a schedule that extends $\mathcal{S}_{i-1}$ on which the starting time of job J_i is maximal. We claim that in $\mathcal{S}$, job J_i is scheduled continuously from its starting time to its deadline. Suppose this is not true, and consider a time point $t < \bar{d}_i$ such that J_i is not being processed in $[t, t+1)$ but is being processed somewhere before. Let $t' < t$ be the first time slot at which J_i is processed.

If $|\mathcal{S}(t)| < m$ then we can obviously remove J_i from $\mathcal{S}(t')$ and add it to $\mathcal{S}(t)$. This would contradict our hypothesis on the starting time of J_i. Hence, $\mathcal{S}$ is full at time t.

We are going to show that the unit of J_i of $\mathcal{S}(t')$ can be exchanged with one unit of some job of $\mathcal{S}(t)$. To do that we evaluate $|\mathcal{S}(t) - \mathcal{S}_{i-1}(t)|$. It is equal to $|\mathcal{S}(t)| - |\mathcal{S}_{i-1}(t)|$ because $\mathcal{S}_{i-1}(t) \subseteq \mathcal{S}(t)$. Hence it equals $m - |\mathcal{S}_{i-1}(t)|$. Recall that $\mathcal{S}_{i-1}$ has a staircase structure so, $|\mathcal{S}_{i-1}(t')| \geq |\mathcal{S}_{i-1}(t)|$. Hence we have

$$\begin{aligned} |\mathcal{S}(t) - \mathcal{S}_{i-1}(t)| &\geq m - |\mathcal{S}_{i-1}(t')| && (3.5)\\ &\geq |\mathcal{S}(t') - \mathcal{S}_{i-1}(t')| && (3.6) \end{aligned}$$

The sets $\mathcal{S}(t) - \mathcal{S}_{i-1}(t)$ and $\mathcal{S}(t') - \mathcal{S}_{i-1}(t')$ are not identical since J_i belongs to the first one and not the second one. Hence there is a job $J_j \in \mathcal{S}(t) - \mathcal{S}_{i-1}(t)$ that does not belong to $\mathcal{S}(t') - \mathcal{S}_{i-1}(t')$. We can perform the exchange between J_i and J_j because the deadlines are not lower than $\bar{d}_i$ (because jobs are scheduled one after the other in non-decreasing order of deadlines). This exchange improves the starting time of J_i; which contradicts our hypothesis.

Let u be the smallest index such that $C_{i-1}^{u+1} + p_i \leq \bar{d}_i$. Algorithm 2 schedules exactly $C_{i-1}^u - (\bar{d}_i - p_i)$ units of J_i between C_{i-1}^{u+1} and C_{i-1}^u (after this date, J_i is scheduled continuously until its deadline). Exactly the same amount of J_i is scheduled over this interval in $\mathcal{S}$. So over each staircase, the same number of units of J_i are scheduled in both $\mathcal{S}$ and $\mathcal{S}_{i-1}$. To transform $\mathcal{S}$ into an extension of $\mathcal{S}_i$, we keep the jobs $J_1, \cdots, J_{i-1}$ fixed, we keep the schedule as it is after time $\bar{d}_i$ and we reschedule over each time interval defined by a staircase the pieces of jobs that execute there. The rescheduling is achieved thanks to Mc Naughton'rule by scheduling first job J_i on the machine whose index is minimal. $\square$

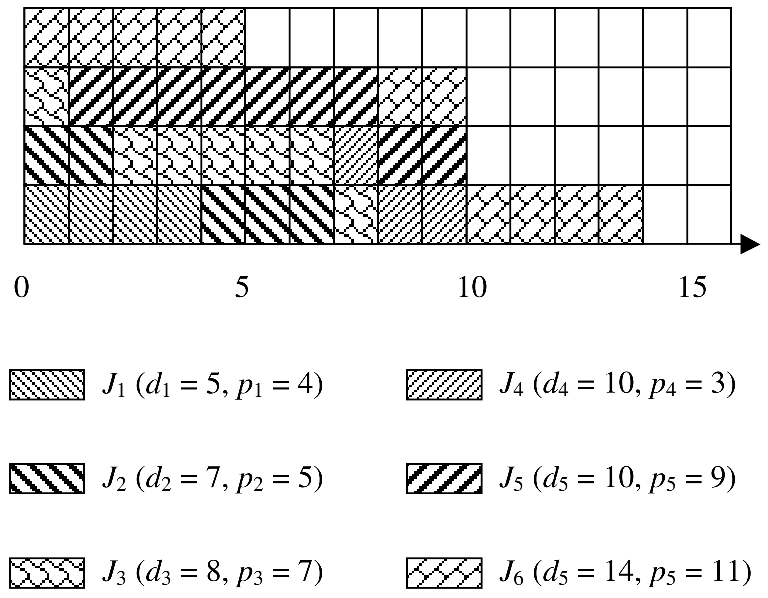

0
5
10
15
J_1 ($d_1 = 5, p_1 = 4$)
J_4 ($d_4 = 10, p_4 = 3$)
J_2 ($d_2 = 7, p_2 = 5$)
J_5 ($d_5 = 10, p_5 = 9$)
J_3 ($d_3 = 8, p_3 = 7$)
J_6 ($d_5 = 14, p_5 = 11$)

Figure 3.1: Scheduling 6 jobs on 4 machines with Algorithm 2

The time complexity of Algorithm 2 is $O(nm)$. This algorithm is not as efficient as Sahni's algorithm that can be implemented in $O(n \log mn)$ steps. However, the fact that machine 1 is always loaded as much as possible, then machine 2, *etc.*, is of prime interest for us. This structure of the schedule will be used later on to precompute the set of possible completion times for the $\sum w_i U_i$ problem with identical processing times.

Recursion Formula for $\sum w_i U_i$

The main interest of Algorithm 2 is that throughout its execution we have a staircase profile, *i.e.*, machines are never idle and the first machine is always more loaded than the second one, *etc.* The weighted number of late jobs problem can be solved by dynamic programming. Informally speaking, the current state is defined by an index i that determines the jobs that remain to be scheduled and by the time points $C^1 \geq C^2 \geq \cdots \geq C^m$ at which machines $1, \cdots, m$ become idle. The current job J_i is either late or on-time. In the latter case it is scheduled according to Algorithm 2.

Definition 3. *For any $C^1 \geq \cdots \geq C^m$ and any $i \leq n$, let $W_i(C^1, \cdots, C^m)$ be the minimum weighted number of late jobs among schedules of $\{J_i, \cdots, J_n\}$ on which machines $1, \cdots, m$ are not available before time $C^1, \cdots, C^m$.*

A shown in Proposition 6 We have a simple recursion formula between variables $W_i(C^1, \cdots, C^m)$.

Proposition 6. *If $C^m + p_i > d_i$ then*

$$W_i(C^1, \cdots, C^m) = W_{i+1}(C^1, \cdots, C^m) + w_i \tag{3.7}$$

otherwise,

$$W_i(C^1, \cdots, C^m) = \min(W_{i+1}(C^1, \cdots, C^m) + w_i, W_{i+1}(C_i^1, \cdots, C_i^m)) \tag{3.8}$$

where C_i^u is completion time of J_i when scheduled according to lines 5–11 of Algorithm 2 where $C^1, \cdots, C^m$ play the role of $C_{i-1}^1, \cdots, Ci - 1^m$.

Proof. If $C^m + p_i > d_i$, J_i cannot be on-time and thus, we have to take into account the "late" cost w_i and the remaining jobs are scheduled by induction. If $C^m + p_i \leq d_i$, then J_i can be either late or on-time. If job J_i is on-time, it is valid to schedule it according to Algorithm 2 because it is the unscheduled job with minimum due-date and because we have a staircase profile ($C^1 \geq \cdots \geq C^m$). The remaining jobs are scheduled by induction. If the job J_i is late, we have to again take into account the "late" cost w_i. □

A Pseudopolynomial Algorithm

Proposition 6 leads to a straight pseudo-polynomial algorithm. First of all, $W_{n+1}(C^1, \cdots, C^m)$ is set to 0 for all possible values of $(C^1, \cdots, C^m)$ such that $\forall j, C_j \leq d_n$ (initialization). Jobs J_i are then taken in decreasing order and the recursion formula is applied on each combination of the C^u values with $C^1 \geq \cdots \geq C^m$. This leads to a time complexity of $O(nmD^m)$, where $D = d_n$ is the maximal due-date, since (1) the initialization runs in $O(D^m)$ and (2) for each job J_i, there are $O(D^m)$ combination of parameters that have to be tested and each time the cost of scheduling J_i according to Algorithm 2 is $O(m)$. A rough analysis of the space complexity required for this dynamic programming algorithm is $O(nD^m)$. However, at each step of the outer loop on i, one only needs the values of W computed at the previous step $(i-1)$, the algorithm can be implemented with 2 arrays of $O(D^m)$ size: one for the current values of W and one for the previous values of W.

Identical Processing Times

Now consider the case where jobs have identical processing times. Our idea is to use the Dynamic Programming algorithm of Section 3.1.3 to schedule these jobs. However, we will see that the completion times of jobs take only a small number (*i.e.*, polynomially bounded for any fixed value of m) of possible values. Let us come back to the deadline scheduling problem and let us analyze what happens when applying Algorithm 2 on n jobs with identical processing times. Proposition 7 exhibit the structure of the completion time on the first machine while Propositions 8 and 9 provide a recursive formula to compute the structure of the completion times on machine u when it is known on $1, \cdots, u-1$. To simplify presentation, we assume that the first job is a fake job with $p_1 = \bar{d}_1 = 0$.

Proposition 7. *Throughout Algorithm 2, C_i^1 equals a due-date modulo p.*

Proof. (By induction on the iteration i of Algorithm 2). If J_i is fully scheduled on machine 1 then C_i^1 is increased of p and the proposition holds. Otherwise the due-date $\bar{d}_i$ has been met, *i.e.*, $C_i^1 = \bar{d}_i$ and the proposition also holds. □

Proposition 8. *If J_i is not completed on machine $u \geq 2$ (i.e., if it is also scheduled on $u+1$, etc.) then C_i^u, the completion time of J_i on u, is equal to C_{i-1}^{u-1}.*

Proof. Immediately comes from the structure of Algorithm 2 (*cf.*, Figure 3.2). □

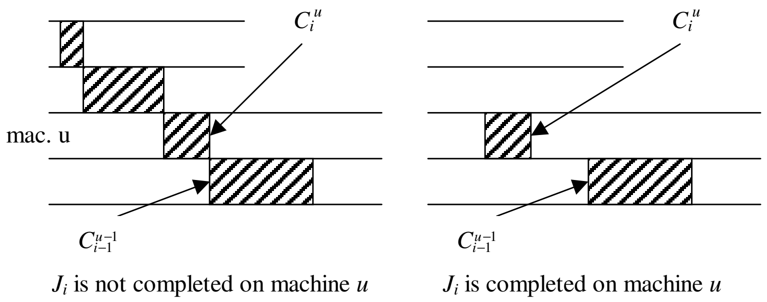

C_i^u
C_i^u
mac. u
C_{i-1}^{u-1}
C_{i-1}^{u-1}
J_i is not completed on machine u
J_i is completed on machine u

Figure 3.2: Illustration of Proposition 8

Proposition 9. *If J_i is completed on machine u (i.e., if it is not scheduled on $u+1$, etc.) then there is a job J_j that is not completed on machine u such that (cf., Figure 3.3)*

$$C_i^u = (i-j)p + C_j^u - \sum_{v=1}^{u-1}(C_i^v - C_j^v) \tag{3.9}$$

Proof. Let J_j $(j < i)$ be the job with maximal index that is not completed on machine u but on a later machine. As shown on Figure 3.3, the $i - j$ jobs $\{J_{j+1}, \cdots, J_i\}$ are scheduled between C_j^1 and C_i^1 on the first machine, between C_j^2 and C_i^2 on the second machine, *etc.* They are all completed before the u-th machine. Since machines are never idle and since the total processing time of jobs is $(i - j)p$, we have

$$\sum_{v=1}^{u} (C_i^v - C_j^v) = (i - j)p \tag{3.10}$$

Hence, (3.9) holds. Now assume that there is no such job J_j then all jobs are completed on u or on a previous machine. So between time 0 and the completion time of J_i on each machine, we had enough space to schedule all jobs, *i.e.*,

$$\sum_{v=1}^{u} C_i^v = ip \tag{3.11}$$

With the convention $p_1 = \bar{d}_1 = 0$, (3.9) holds. $\square$

Note that in the previous proposition, C_j^u is also equal to the completion time of some other job on machine $u - 1$ (*cf.*, Proposition 8).

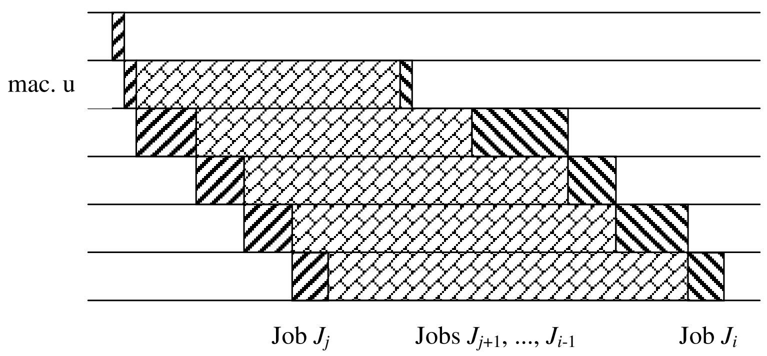


Figure 3.3: Jobs between J_j and J_i

Now let us come back to the $\sum w_i U_i$ problem and let us compute machine per machine some sets of time points $\mathcal{T}_1, \cdots, \mathcal{T}_m$ that include all possible completion times. Due to Proposition 7, we have

$$\mathcal{T}_1 = \{d_i + \lambda p, 1 \leq i \leq n, 1 \leq \lambda \leq n\} \tag{3.12}$$

Thanks to Proposition 9, we can compute recursively $\mathcal{T}_2, \mathcal{T}_3$, *etc.*. Indeed, a value $t \in \mathcal{T}_u$ is obtained by picking

- $u - 1$ values $(\alpha_1, \cdots, \alpha_{u-1})$ respectively in $\mathcal{T}_1, \cdots, \mathcal{T}_{u-1}$ (these values play the role of the completion times of J_j on the $u - 1$ first machines in Proposition 9),

- 1 value in $\mathcal{T}_{u-1}$ that corresponds to the completion time of J_j on the machine u (as we have noted, this value is also equal to the completion time of some other job on machine $u - 1$),

- $u - 1$ values $(\beta_1, \cdots, \beta_{u-1})$ respectively in $\mathcal{T}_1, \cdots, \mathcal{T}_{u-1}$ that play the role of the completion times of J_i on the $u - 1$ first machines,

- an integer $\lambda \in \{1, \cdots, n\}$ that would equal $j - i$ in Proposition 9 and by computing

$$t = \alpha + \lambda p - \sum_{v=1}^{u-1} (\beta_v - \alpha_v) \tag{3.13}$$

It is easy to get a rough upper bound of $|\mathcal{T}_u|$:

$$|\mathcal{T}_u| \leq n|\mathcal{T}_{u-1}| \prod_{v=1}^{u-1} |\mathcal{T}_v|^2 \leq n|\mathcal{T}_{u-1}|^{2u-1} \tag{3.14}$$

The key point is that there are $O(n^{f(m)})$ values in $\mathcal{T}_m$ where f is some function that depends only of m (*e.g.*, $f(m) = 2^m m!$ is a valid upper bound). So, for any fixed value of m, we have a polynomial number of possible completion times. Once they are computed, we can restrict the dynamic programming search to those values of the completion times and we obtain a strongly polynomial algorithm for $Pm|pmtn, p_i = p|\sum w_i U_i$.

3.2 A Non-Preemptive Problem, $P_m|r_i, p_i = p|\sum w_i U_i$

In this section, we show that, in the non-preemptive case, most of the results described in Chapter 2 can be extended to the situation where m parallel identical machines are available.

$P|r_i|\sum f_i(C_i)$ consists of finding a set of completion times C_i for each job J_i such that (1) jobs start after their release date, *i.e.*, $\forall i, C_i - p_i \geq r_i$, (2) no more than m machines are used at any time t, *i.e.*, $\forall t, |\{J_i : C_i - p_i \leq t < C_i\}| \leq m$, and (3) the objective function $\sum_i f_i(C_i)$ is minimal. In the following, a schedule meeting (1) and (2) is said to be feasible.

We study the special case where processing times are equal. As shown in the literature, this assumption sometimes allows to exhibit polynomial time algorithms for problems that are NP-hard in the general case:

- Scheduling identical jobs on uniform parallel machines (*i.e.*, on machines that do not run at the same speed), is polynomial when release dates are equal and when the scheduling criteria is non-decreasing in the job completion times (see for instance [86]). In the case of distinct release dates, Dessouky, Lageweg, Lenstra and van de Velde show ([86]) that both the problem of minimizing makespan and the problem of minimizing the sum of the completion times (for a fixed number m of uniform machines) are polynomial.

- Simons ([201]) provides a polynomial algorithm to minimize the sum of the completion times when processing times are equal ($P|p_i = p, r_i|\sum C_i$) while the simple problem ($1|r_i|\sum C_i$) is NP-Hard [147].

We show that, for any ordered objective function F (Definition 2), $Pm|r_i, p_i = p|F$ can be solved in polynomial time. Hence, $Pm|p_j = p, r_j|\sum_j w_j C_j$, $Pm|p_j = p, r_j|\sum_j T_j$ and $Pm|p_j = p, r_j|\sum_j T_j$ are in P.

We extend a dominance property (Section 3.2) initially introduced for the single machine case and we describe the variables of the dynamic programming algorithm and the Proposition that links them together (Section 3.2). The dynamic programming algorithm itself is described in Section 3.2.

Dominance Property

As for the one-machine case, we can restrict our search to active schedules. Proposition 10 provides a characterization of the time points at which jobs start and end on active schedules.

Proposition 10. *In active schedules, start and completion times belong to*

$$\Theta = \{t|\exists r_i, \exists l \in \{0, \ldots, n\}, t = r_i + lp\}.$$

Proof. Same proof as Proposition 3. □

Variables Definition and Decomposition Scheme

Recall that late jobs can be scheduled arbitrary late. Hence, there is no point to assign a starting time to late jobs. In the following, when we refer to a schedule $\mathcal{H}$ of some subset of jobs O, it does not mean that all jobs in O are in the schedule (the on-time jobs only are in the schedule). So the "cost" of $\mathcal{H}$ with respect to O is

- the sum, over all jobs $J_i \in O$ scheduled in $\mathcal{H}$, of $f_i(C_i)$
- plus the sum, over all other jobs $J_i \in O$ not scheduled in $\mathcal{H}$, of ω_i.

Roughly speaking, the decomposition scheme for the single machine case (Proposition 4) relies on a particular time point $t_k \in [s, e)$ that allows us to split the problem into two sub-problems (between s and t_k and between t_k and e). The same scheme is kept here, however s and e are replaced by two vectors σ and ϵ that represent some resource profiles. We introduce more formally this notion.

Definition 4. *A resource profile ξ is a vector $(\xi_1, \xi_2, \ldots, \xi_m)$ such that $\xi_1 \leq \xi_2 \leq \ldots \leq \xi_m$ and $\xi_m - \xi_1 \leq p$. In the following, Ξ denotes the set of resource profiles ξ such that $\forall i, \xi_i \in \Theta$.*

Intuitive meaning of resource profiles. A resource profile $\xi = (\xi_1, \xi_2, \ldots, \xi_m)$ represents the state of the resource at some time point ξ_1 (in practice this time point always matches the starting time or the completion time of a job). If q machines are idle at time ξ_1 then the q first components of the resource profile equal ξ_1. The other components are the completion times of the $m - q$ jobs that are being processed at time ξ_1. Since the processing time of jobs is p, we have $\forall i, \xi_i - \xi_1 \leq p$. The state of the resource is considered from a global point of view and thus, the i^{th} component of the resource profile is not systematically related to the i^{th} machine. In the following, some "left" resource profiles σ are used to state that no machine is available before σ_1, one machine is available between σ_1 and σ_2; two machines are available between σ_2 and σ_3, *etc.* Conversely, some "right" resource profiles ϵ are used to state that no machine is available after ϵ_m, one machine is available between ϵ_{m-1} and ϵ_m; two machines are available between ϵ_{m-2} and ϵ_{m-1}, *etc.* The resource profiles σ and ϵ allow us to determine the exact amount of resource that is available at each time point in $[\sigma_1, \epsilon_m]$.

Definition 5. *Given two resource profiles ξ and μ, $\xi \ll \mu$ if and only if for any index i in $\{1, \ldots, m\}, \xi_i \leq \mu_i$.*

Notice that $\ll$ defines a partial order on resource profiles. We introduce a technical proposition that will be used later on.

Proposition 11. *Given two resource profiles ξ and μ, $\xi \ll \mu$ if and only if for any time point t, $|\{i : t < \xi_i\}| + |\{i : \mu_i \leq t\}| \leq m$.*

Proof. **Sufficient condition**. (By contradiction.) Let k be the first index such that $\mu_k < \xi_k$. Let us compute $|\{i : t < \xi_i\}| + |\{i : \mu_i \leq t\}|$ for $t = \mu_k$. It is equal to $|\{i : \mu_k < \xi_i\}|$ plus $|\{i : \mu_i \leq \mu_k\}|$; which is greater than or equal to $(m - k + 1) + k > m$. This contradicts our hypothesis.
Necessary condition. Let t be any time point. In the following, $\delta(P)$ denotes the binary variable that equals 1 if the condition P holds, 0 otherwise. Notice that for any value of i, $\delta(t < \xi_i) + \delta(\mu_i \leq t) \leq 1$ because $\xi_i \leq \mu_i$. Hence,

$$|\{i : t < \xi_i\}| + |\{i : \mu_i \leq t\}| = \sum_{i=1}^{m} (\delta(t < \xi_i) + \delta(\mu_i \leq t)) \leq m.$$

□

We now define the variables of the dynamic programming algorithm. For any integer $k \leq n$, for any resource profiles σ and ϵ ($\sigma \ll \epsilon$), let $F_k(\sigma, \epsilon)$ be the minimal cost over all feasible schedules of the jobs in $U_k(\sigma_m - p, \epsilon_1)$ such that

- starting times belong to Θ,

- the number of machines available at time t to schedule the jobs in the set $U_k(\sigma_m - p, \epsilon_1)$ is $m - |\{i : t < \sigma_i\}| - |\{i : \epsilon_i \leq t\}|$.

Notice that given our definition, $F_0(\sigma, \epsilon)$ is equal to 0.

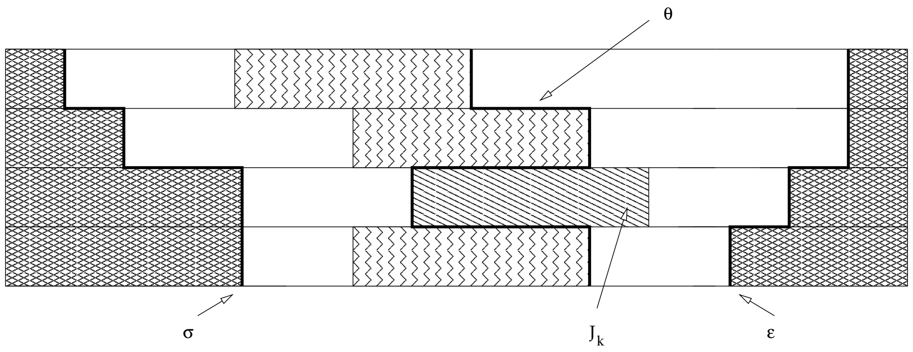


Figure 3.4: Resource profiles

Proposition 12. *Let $k \in [1, n]$ and let $\sigma \ll \epsilon$ be two resource profiles. If $r_k \notin [\sigma_m - p, \epsilon_1)$, $F_k(\sigma, \epsilon) = F_{k-1}(\sigma, \epsilon)$. If $r_k \in [\sigma_m - p, \epsilon_1)$, $F_k(\sigma, \epsilon)$ is equal to $\min(\omega_k + F_{k-1}(\sigma, \epsilon), F')$; where*

$$F' = \min_{\substack{\theta \in \Xi \\ r_k \leq \theta_1 \\ \sigma \ll \theta \\ \theta' = (\theta_2, \ldots, \theta_m, \theta_1 + p) \\ \theta' \ll \epsilon}} F_{k-1}(\sigma, \theta) + F_{k-1}(\theta', \epsilon) + f_k(\theta_1 + p)$$

(If F' is undefined, assume that $F' = \infty$.)

Notice that in the above formula the value θ' is derived from θ. Figure 3.4 provides an illustration of this proposition. Proposition 12 basically states that the optimum schedule for k, σ, ϵ can be computed by trying all possible resource profiles θ of Ξ that are "between" the resource profiles σ and ϵ. For each candidate resource profile θ, the job J_k starts at θ_1.

Proof. First notice that, θ' is a resource profile because $\theta \in \Xi$. Hence the use of $F_{k-1}(\theta', \epsilon)$ is correct. If $r_k \notin [\sigma_m - p, \epsilon_1)$ the proposition obviously holds because $U_k(\sigma_m - p, \epsilon_1) = U_{k-1}(\sigma_m - p, \epsilon_1)$. We now consider the case where $r_k \in [\sigma_m - p, \epsilon_1)$.
We first prove that $\omega_k + F_{k-1}(\sigma, \epsilon) \geq F_k(\sigma, \epsilon)$ and that $F' \geq F_k(\sigma, \epsilon)$. The first claim is obvious (consider the schedule that realizes $F_{k-1}(\sigma, \epsilon)$, "add" J_k and put it late). Now, let $\theta \in \Xi$ be the resource profile that realizes F'. There is a schedule $\mathcal{H}_1$ that realizes $F_{k-1}(\sigma, \theta)$ and a schedule $\mathcal{H}_2$ that realizes $F_{k-1}(\theta', \epsilon)$. Notice that any job in $U_{k-1}(\sigma_m - p, \epsilon_1)$ is either late or scheduled in $\mathcal{H}_1$ or in $\mathcal{H}_2$. Consider the schedule $\mathcal{H}$ build as follows: schedule J_k at time θ_1 and all other jobs in $U_k(\sigma_m - p, \epsilon_1)$ at the time they were scheduled on $\mathcal{H}_1$ or on $\mathcal{H}_2$. Let us prove that $\mathcal{H}$ is a feasible schedule of $U_k(\sigma_m - p, \epsilon_1)$. Since $r_k \leq \theta_1$ and since $\mathcal{H}_1$ and $\mathcal{H}_2$ are feasible, all jobs are scheduled after their release date. Moreover, we claim that the m-machine constraint holds.
Let t be any time point. The number of machines used by the jobs scheduled on $\mathcal{H}_1$ is upper bounded by $m - |\{i : t < \sigma_i\}| - |\{i : \theta_i \leq t\}|$. The number of machines used by the jobs scheduled on $\mathcal{H}_2$ is upper bounded by $m - |\{i : t < \theta'_i\}| - |\{i : \epsilon_i \leq t\}|$. Finally, J_k uses a machine at time t if and only if $t \in [\theta_1, \theta_1 + p)$. The resource constraint is satisfied at time t if the sum of the upper bounds is lower than or equal to $m - |\{i : t < \sigma_i\}| - |\{i : \epsilon_i \leq t\}|$, *i.e.*, if the following expression is lower than

or equal to 0.

$$\begin{aligned}
& m - |\{i : t < \theta'_i\}| - |\{i : \theta_i \leq t\}| + \delta(\theta_1 \leq t < \theta_1 + p) \\
= \; & m - \sum_{i=1}^{m} \delta(t < \theta'_i) - \sum_{i=1}^{m} \delta(\theta_i \leq t) + \delta(\theta_1 \leq t < \theta_1 + p) \\
= \; & m - \left(\sum_{i=2}^{m} \delta(t < \theta_i) + \delta(t < \theta_1 + p) \right) - \sum_{i=1}^{m} \delta(\theta_i \leq t) + \delta(\theta_1 \leq t < \theta_1 + p) \\
= \; & m - \sum_{i=2}^{m} (\delta(t < \theta_i) + \delta(\theta_i \leq t)) - \delta(t < \theta_1 + p) - \delta(\theta_1 \leq t) + \delta(\theta_1 \leq t < \theta_1 + p) \\
= \; & 1 - \delta(t < \theta_1 + p) - \delta(\theta_1 \leq t) + \delta(\theta_1 \leq t < \theta_1 + p) \leq 0
\end{aligned}$$

We have proven that $\mathcal{H}$ is a feasible schedule of $U_{k-1}(\sigma_m - p, \epsilon_1)$. On top of that, starting times obviously belong to Θ. The cost of $\mathcal{H}$ is exactly F' and hence, $F' \geq F_k(\sigma, \epsilon)$.
We now prove that $\min(\omega_k + F_{k-1}(\sigma, \epsilon), F') \leq F_k(\sigma, \epsilon)$. Consider a schedule that realizes $F_k(\sigma, \epsilon)$ and let O be the set of jobs scheduled in this schedule (jobs in $O - U_k(s - p, e)$ are late). Among all schedules (1) that realize $F_k(\sigma, \epsilon)$ and (2) in which the same set of jobs O are scheduled, consider the schedule $\mathcal{H}$ that lexicographically minimizes the vector made of completion times of jobs in O. (The completion time of the job in O, with the smallest index, is minimum, then the completion time of the job in O, with the second smallest index, *etc.*). The job J_k is either late or on-time on $\mathcal{H}$.
If J_k is late then $\mathcal{H}$ is it is also a schedule of $U_{k-1}(\sigma_m - p, \epsilon_1)$ that could fit between σ and ϵ. Its cost cost is exactly $F_k(\sigma, \epsilon) - \omega_k$ (the late cost ω_k is removed since J_k is not considered any longer). Hence $F_{k-1}(\sigma, \epsilon) \leq F_k(\sigma, \epsilon)$.
Now assume that J_k is on-time and let t_k be its starting time on $\mathcal{H}$. The proof works as follows. We first show that on $\mathcal{H}$, jobs with a release date lower than or equal to t_k are either late or start before or at t_k. We then exhibit a resource profile $\theta \in \Xi$ such that (1) $\theta_1 = t_k$, (2) $\sigma \ll \theta$, (3) $\theta' = (\theta_2, \ldots, \theta_m, \theta_1 + p) \ll \epsilon$. We then conclude the proof.
<u>In $\mathcal{H}$, jobs J_i with $r_i \leq t_k$ are late or start before or at t_k.</u> Suppose that there is an on-time job J_i with $r_i \leq t_k$ that is executed at time $t_i > t_k$ on $\mathcal{H}$. Let $\mathcal{H}'$ be the schedule obtained from $\mathcal{H}$ by exchanging the jobs J_i and J_k. $\mathcal{H}'$ is better than $\mathcal{H}$ (*cf.* proof of the second part of Proposition 4). This contradicts our hypothesis on $\mathcal{H}$.
<u>Definition of θ.</u> Let τ the vector build component per component as follows: The first component of τ is t_k, the time at which J_k starts. The following components of τ are the end times on $\mathcal{H}$ of the jobs (except J_k) that start before or at t_k and end strictly after t_k. The following components are the values σ_i that are strictly greater than t_k. Since the resource constraint holds at time t for $\mathcal{H}$, it is easy to prove that the dimension of τ is lower than or equal to m. The vector τ is extended to a vector τ' of dimension m by adding a sufficient number of times a component t_k. Let θ be the vector obtained from τ' by sorting its components in increasing order.
<u>θ belongs to Ξ.</u> Consider a component θ_j of θ. Either it is the end time of a job and then $t_k < \theta_j \leq t_k + p$) or it is a $\sigma_i > t_k$ value (and then $t_k < \theta_j \leq t_k + p$ otherwise t_k would be strictly lower than σ_1 and hence no machine would be available at time t_k) or it is equal to t_k. Hence, all components belong to the interval $[t_k, t_k + p]$, as a consequence $\theta_m - \theta_1 \leq p$. We have proven that θ is a resource profile. It is also easy to verify that all components of θ belong to Θ and hence $\theta \in \Xi$. On top of that, it is obvious that $\theta_1 = t_k$. The proof that $\sigma \ll \theta$ is also immediate given the definition of θ.
<u>$\theta' = (\theta_2, \ldots, \theta_m, \theta_1 + p) \ll \epsilon$.</u> The fact that θ' is a resource profile comes immediately from $\theta \in \Xi$. Suppose that the relation $\theta' \ll \epsilon$ does not hold. Then, according to Proposition 11, there is a time point t such that $|\{i : t < \theta'_i\}| + |\{i : \epsilon_i \leq t\}| > m$. Recall that $\epsilon_1 \geq t_k$ otherwise no machine would be available at the time point where J_k ends. Hence, if $t < t_k$, $|\{i : \epsilon_i \leq t\}| = 0$ and consequently, $|\{i : t < \theta'_i\}| > m$; which contradicts the fact that θ' is a vector of dimension m. As a consequence, we have $t_k \leq t$. Let O be the set of jobs that start before or at t_k and end strictly after t_k. The components of θ', are either the completion times of the jobs in O or the σ_i values that are strictly greater than t_k or are equal to t_k. Hence, the number of components of θ' that are strictly greater than t is equal to the sum of (1) the number of jobs in O that end strictly after time t and of (2) the

number of components of σ_i that are strictly greater than t. Since $t_k \le t$, the jobs in O all start before t. Hence, the total number of jobs N_t that start before or at t and end strictly after t plus the number of components of σ_i that are strictly greater than t, is greater than or equal to $|\{i : t < \theta'_i\}|$. Hence, $|\{i : t < \theta'_i\}| + |\{i : \epsilon_i \le t\}| > m$ leads to, $N_t + |\{i : t < \sigma_i\}| + |\{i : \epsilon_i \le t\}| > m$. This contradicts the fact that the resource constraint is met at time t on $\mathcal{H}$.
Conclusion. The cost of $\mathcal{H}$, restricted to the jobs with a release date lower than or equal to t_k (except J_k) is not lower than $F_{k-1}(\sigma, \theta)$. Similarly, the cost of $\mathcal{H}$ restricted to the jobs with a release date greater t_k is not lower than $F_{k-1}(\theta', \epsilon)$. Hence the total cost of $\mathcal{H}$ ($F_k(\sigma, \epsilon)$), is greater than or equal to $F_{k-1}(\sigma, \theta) + F_{k-1}(\theta', \epsilon) + f_k(\theta_1 + p)$. □

A Dynamic Programming Algorithm

The optimum is exactly $F_n((\min_{t\in\Theta} t, \ldots, \min_{t\in\Theta} t), (\max_{t\in\Theta} t, \ldots, \max_{t\in\Theta} t))$. Thanks to Proposition 12, we have a straight dynamic programming algorithm to compute this value. The relevant values for σ and ϵ are exactly the vectors in Ξ. We claim that there are $O(n^2 n^{m-1}) = O(n^{m+1})$ relevant resource profiles. Indeed, there are n^2 possible values for the first component and once it is fixed there are only n possible choices for the $m-1$ remaining ones (because of the structure of Θ and because the difference between the m-th component and the first one is upper bounded by p). This means that there are $O(n^{2m+2})$ relevant pairs (σ, ϵ). The values of $F_k(\sigma, \epsilon)$ are stored in a multi-dimensional array of size $O(n^{2m+3})$ (n possible values for k, n^{m+1} possible values for σ and n^{m+1} possible values for ϵ). Our algorithm then works as follows.

- In the initialization phase, $F_0(\sigma, \epsilon)$ is set to 0 for any values σ, ϵ in Ξ such that $\sigma \ll \epsilon$.
- We then iterate from $k = 1$ to $k = n$. Each time, F_k is computed for all the possible values of the parameters thanks to the formula of Proposition 12, and to the values of F_{k-1} computed at the previous step.

Before analyzing the complexity of the overall algorithm, remark that one can generate easily all possible resource profiles θ between σ and ϵ (*i.e.*, $\sigma \ll \theta \ll \epsilon$) in $O(n^{m+1})$ steps. Indeed, there are $O(n^2)$ possible values $\theta_1 \in \Theta \cap [\sigma_1, \epsilon_1]$. The other components of θ belong to $\Theta \cap [\theta_1, \theta_1 + p]$. There are only $O(n)$ values in this set. Components θ_i are generated, one after another; each time a test verifying that $\sigma_i \le \theta_i \le \epsilon_i$ being performed in constant time.
In the initialization phase, $O(n^{2m+2})$ pairs ($\sigma \ll \epsilon$) are generated. Afterwards, for each value of k, $O(n^{2m+2})$ values of F_k have to be computed. For each of them, $O(n^{m+1})$ resource profiles θ are generated with $\sigma \ll \theta \ll \epsilon$. A minimum among $O(n^{m+1})$ terms is computed. This leads to an overall time complexity of $O(n^{3m+4})$. A rough analysis of the space complexity leads to an $O(n^{2m+3})$ bound but since, at each step of the outer loop on k, one only needs the values of F computed at the previous step $(k-1)$, the algorithm can be implemented with 2 arrays of $O(n^{2m+2})$ size: one for the current values of F and one for the previous values of F.
Notice that we can perform a backward computation on the values $F_k(\sigma, \epsilon)$ to recover the optimum schedule. Indeed, for all relevant values of k, σ and ϵ, it is easy to identify which resource profile θ is the best (according to the fundamental recursion formula of Proposition 8). Since the starting time of J_k is the first component θ_1 of θ, we can recover the starting times of all jobs at once.

Open Problems

We have shown that scheduling equal length jobs on a fixed number of parallel identical machines is a polynomial problem for some objective functions. We have established that several open scheduling problems such as $Pm|p_j = p, r_j|\sum_j w_j U_j$, $Pm|p_j = p, r_j|\sum_j w_j C_j$ or $Pm|p_j = p, r_j|\sum_j T_j$ are polynomial. The weighted total tardiness problem is still an open problem (even for $m = 1$).

3.3 UET Open-Shops, $Om|p_{ij} = 1, r_i|\sum w_i U_i$

We study unit execution time open-shops with integer release dates. As shown below, this problem is strongly related to the problem of scheduling identical jobs on parallel machines. We show that the minimum weighted number of late jobs can be computed in polynomial time by dynamic programming. The complexity status of the corresponding problem $Om|p_{ij} = 1, r_i| \sum w_i U_i$ was unknown before.

3.3.1 Problem Definition

We have n jobs $\{J_1, \ldots, J_n\}$ to be scheduled on m machines $\{M_1, \ldots, M_m\}$. Each job J_i consists of m operations $\{O_{i1}, \ldots, O_{im}\}$ that cannot overlap in time. Operation O_{ij} has to be processed on machine M_j during p_{ij} time units. A release date r_i, a due date d_i and a weight w_i are associated with each job J_i. In the following, all data are integer. This environment defines an open-shop scheduling problem. We focus on the problem of minimizing, in the non-preemptive case, the weighted number of late jobs; a job J_i is late if and only if it is completed after its due date d_i.

This problem, denoted as $O|r_i| \sum w_i U_i$ in the standard scheduling terminology (U stands for Unit penalty per late job), is NP-hard, even if release dates are equal. We study the special case $O|p_{ij} = 1, r_i| \sum w_i U_i$ where jobs have Unit Execution Time (UET). In this situation, operations of the same job can be exchanged and the constraint "Operation O_{ij} has to be processed on machine M_j" can be replaced by "Operations of the same job have to be processed on distinct machines". We refer to [38, 47, 112, 135, 209, 210] for extended and up-to-date complexity results on UET shop scheduling problems.

- Liu and Bulfin have shown that when both release dates and weights are equal, the problem $O|p_{ij} = 1| \sum U_i$ is polynomial. An $O(mn^2)$ algorithm is described in [156].

- Brucker, Jurisch, Tautenhahn and Werner have studied the special case where release dates are equal and where the number of machines is fixed $Om|p_{ij} = 1| \sum w_i U_i$. It is shown in [45] that the problem can be solved in $O(n^2 m^{m+1})$, which is $O(n^2)$ for any fixed value of m.

- Different release dates make the problem harder. Gladky [110] has shown that $O|r_i, p_{ij} = 1| \sum w_i U_i$ is NP-hard and Kravchenko [133] has extended the NP-hardness proof to $O|r_i, p_{ij} = 1| \sum U_i$.

- Galambos and Woeginger [107] have shown that for two machines the problem with release dates $O2|p_{ij} = 1, r_i| \sum w_i U_i$ can be solved in $O(n^4 \log n)$.

These results leave open the status of the problem $Om|p_{ij} = 1, r_i| \sum w_i U_i$, for each value of m greater than or equal to 3. We show that for any fixed value of m, the problem can be solved in $O(n^{4m^2 - m + 10})$. In §3.3.2, we recall a well-known transformation of the UET open shop into a preemptive scheduling problem on identical parallel machines. We introduce in §3.3.3 some dominance properties on this problem. In §3.3.4, we define the variables of the dynamic programming algorithm and we present the fundamental recursion formula. The algorithm is described in §3.3.5. Finally, we draw some conclusions in §3.3.6.

3.3.2 Transformation

For the sake of completeness, we briefly recall the main result of [43] concerning the transformation of UET open-shops into a identical parallel machine scheduling problem. We refer to [210] for recent developments on the strong links that exist between shop problems and parallel machine scheduling problems. In the following, F denotes a regular function of the completion times of the jobs.

We can assume that $m < n$, because when $n \le m$, the optimum schedule can be computed as follows: Schedule each operation O_{ij} at time $t_{ij} = r_i + j - 1$ on the machine M_{u+1} with $u \cong i + t_{ij} [mod\ m]$. It is easy to verify that operations of the same job are processed on distinct machines at consecutive (and distinct) time points. Now consider two operations O_{ij} and $O_{i'j'}$ and suppose that they are processed at the same time (*i.e.*, $t_{ij} = t_{i'j'}$) on the same machine, then $i \cong i'$ $[mod\ m]$. Since i and i' are at most equal to $n \le m$, we have $i = i'$ and $j = j'$, which contradicts our hypothesis. Since each job is completed at its earliest possible completion time, and since F is regular, the schedule is optimal.

An instance of the UET open-shop problem is transformed into an instance of a preemptive parallel machine problem $Pm|pmtn, p_i = m, r_i|F$ as follows. The number of machines m remains the same, and each job J_i of the open-shop is transformed into a job of the parallel machine problem with the same release date and with a processing time $p_i = m$. We keep the term "operation" for the parallel machine problem. It then denotes a unitary piece of a job. It is obvious that a feasible schedule of the open-shop is also a feasible schedule of the parallel machine problem. Brucker, Jurisch and Jurisch [43] have shown that the converse proposition holds. They introduce an algorithm running in $O(mn \log^2(mn))$ that allocates a proper machine to each operation.

Since n release dates have to be coded and because m is fixed, the size of the problem is at least n (and does not depend of m). Therefore the transformation of Brucker, Jurisch and Jurisch [43] is polynomial in the size of the problem. Hence, UET open-shop problems are polynomial if one can preemptively schedule jobs of length m on m identical parallel machines in polynomial time. Unfortunately the complexity of $Pm|pmtn, p_i = m, r_i|F$ is still unknown, even for simple criteria such as the total completion time or the total number of late jobs. We have described (Chapter 2) an $O(n^{10})$ dynamic programming algorithm for minimizing, in the preemptive case, the weighted number of late jobs (with integer release dates) on a single machine when processing times are equal $1|pmtn, p_i = p, r_i|\sum w_i U_i$. However, it does not seem that this algorithm can be adapted to parallel machines. When preemption is not allowed, the problem of scheduling equal length jobs (with integer release dates) on identical parallel machines is studied and solved in Section 3.2. It is shown that for several criteria F, including the weighted sum of the completion times and the sum of the tardiness, the corresponding non-preemptive problem $Pm|p_i = p, r_i|F$ can be solved in $O(n^{3m+4})$. Again, a generalization to the preemptive case of these algorithms, or of other algorithms, initially designed for the non-preemptive scheduling of equal length jobs [53, 55, 86, 109, 201], seems unlikely.

From now on, we consider the parallel machine scheduling problem $Pm|pmtn, p_i = m, r_i|\sum w_i U_i$. This problem is "easier" than the original open-shop because it's not necessary to precisely allocate machines to operations. In the search for a solution, we only have to check that no more than m operations are processed simultaneously. This is the key of the decomposition procedure presented in §3.3.4.

3.3.3 Dominance Properties

Since when a job is late it can be arbitrarily late, we look for schedules in which a maximum weighted number of jobs are scheduled between their release date and their due date and we do not schedule the late jobs. So, in the following, jobs that are scheduled at some time point, are early jobs.

From now on, we suppose that jobs are sorted in non-decreasing order of due-dates, *i.e.*, $d_1 \leq d_2 \leq \ldots \leq d_n$. We first define a set of time points Θ at which jobs start and are completed in some dominant schedules (*cf.*, Proposition 13).

Definition 6. *Θ is the set of time points t for which there exists a release date r_i such that $r_i \leq t \leq r_i + n + m$.*

Notice that there are at most $n(n + m + 1) = O(n^2)$ values in Θ.

Proposition 13. *Start times and completion times of operations in active schedules belong to Θ.*

Proof. A schedule is said to be active if no operation can start earlier without delaying another operation or violating its release date. Let $\mathcal{S}$ be such a schedule, let O_{jk} be an operation starting at t_{jk} in $\mathcal{S}$ and let r_i be the largest release date earlier than or equal to t_{jk}.

If there are at most m time points in $[r_i, t_{jk})$ where at least a machine is idle then, because at most $m - 1$ operations of J_j are processed in $[r_i, t_{jk})$, there is a time point x in $[r_i, t_{jk})$ where a machine is idle and where no operation of J_j is processed. Since $r_j \leq r_i$, the first operation of J_j that is processed after x can be shifted to time x. This contradicts the fact that $\mathcal{S}$ is active.

Hence, between r_i and t_{jk}, there are at most $m - 1$ time points where a machine is idle. Since there are at most mn operations to schedule (n jobs of length m) and since at most m machines are available to process these operations, the distance between r_i and $t_{jk} - (m - 1)$ is upper bounded by $(mn)/m = n$. Hence, $t_{jk} \leq r_i + n + m - 1$. □

Since for any regular criteria, any schedule can be changed into an active one, Proposition 13 induces a simple dominance criterion: There is an optimal schedule in which starting times and completion times belong to Θ. We refine this dominance property.

Definition 7. *A schedule is well-ordered in a time interval $[s, e]$ if and only if*

- *starting times and completion times of operations belong to Θ and,*
- *for any operation O_{ip}, processed at $t_{ip} \in [s, e)$, at most $(m-1)m+1$ jobs J_j, with $r_j \leq t_{ip}$ and $j \leq i$, are completed after time t_{ip}.*

A schedule is well-ordered if and only if it is well-ordered in any interval.

Proposition 14. *There is an optimal well-ordered schedule.*

Proof. Let O be an optimum set of jobs, *i.e.*, a set of early jobs in an optimal schedule and let $\mathcal{S}$ be the schedule of the jobs in O, that lexicographically minimizes the vector U defined as follows. In (3.15), t_{ij} is the starting time of O_{ij} if $J_i \in O$ and $t_{ij} = 0$ otherwise (late jobs are not considered).

$$U = (t_{1\,1}, \ldots, t_{1m}, t_{2\,1}, \ldots, t_{2m}, \ldots, t_{n1}, \ldots, t_{nm}) \tag{3.15}$$

U is lexicographically minimum, hence, $\mathcal{S}$ is active. As a consequence, operations start and are completed at time points in Θ (*cf.*, Proposition 13). Let J_i be any job in O and let O_{ip} be any operation of this job. Let V be the set of jobs that are processed at time t_{ip}, (*i.e.*, one of their operations is processed at time t_{ip}). For each job $J_j \in V$, there are at most $m-1$ time points t greater than t_{ip} at which J_j is processed. Hence, there are $(m-1)m$ time points t greater than t_{ip} such that a job processed at t_{ip} is also processed at time t. Suppose that more than $(m-1)m$ jobs $J_j \in O$, such that $r_j \leq t_{ip}$ and $j < i$, are completed after time t_{ip}. At least one operation of one of these jobs, say O_{jv}, is processed after t_{ip} when no job of V is. Let us modify the schedule $\mathcal{S}$ by exchanging O_{ip} and O_{jv} (the exchange is possible because $j < i \Rightarrow d_j \leq d_i$). The resulting schedule, after a renumbering of operations, is still optimal. Moreover, the schedule is better for the lexicographical order than $\mathcal{S}$; which contradicts our hypothesis on $\mathcal{S}$. We have proven that at most $(m-1)m$ jobs J_j with $r_j \leq t_{ip}$ and $j < i$, are completed strictly after time t_{ip}. Consequently, at most $(m-1)m+1$ jobs J_j with $r_j \leq t_{ip}$ and $j \leq i$, are completed strictly after time t_{ip}. $\square$

3.3.4 Variables Definitions and Fundamental Recursion Formula

We first introduce the remaining processing time (rpt for short) functions that describe the "state" of a schedule at some time point (§3.3.4). They allow us to define the dynamic programming variables (§3.3.4) that are used in the decomposition scheme (§3.3.4 and §3.3.4).

RPT-Functions

Definition 8. *The rpt-function of a schedule $\mathcal{S}$ at time t is the function that associates to each job J_i with $r_i < t$, the number of operations of J_i processed after or at t.*

Of course, for any rpt-function τ of a schedule at time t, we have $0 \leq \tau(J_i) \leq m$ and if $d_i \leq t$ then $\tau(J_i) = 0$ (because late jobs do not appear in the schedule).

Dynamic Variables

We consider some sub-problems defined by five parameters $k, s, e, \sigma, \varepsilon$, where σ and ε are some functions respectively defined over $\{J_i, i \leq k, r_i < s\}$ and $\{J_i, i \leq k, r_i < e\}$ that take their values in $[0, m]$. The sub-problem consists of finding a "feasible" (Definition 9) sub-schedule of the k-first jobs whose "weight" (Definition 10) in the time-interval $[s, e)$ is maximum.

Definition 9. *A schedule $\mathcal{S}$ of the k first jobs is feasible for $k, s, e, \sigma, \varepsilon$ if and only if (1) it is well-ordered and feasible in $[s, e)$ (at most m operations are processed in parallel) and (2) its rpt-functions at s and e restricted to the k first jobs are σ and ε.*

Definition 10. *Let $J(\mathcal{S}, s, e)$ be the set of the jobs available after or at s ($s \leq r_i$) that are fully processed (i.e., m of their operations are processed) in the interval $[s, e)$ in $\mathcal{S}$. The weight of a schedule $\mathcal{S}$ in $[s, e)$ is $\sum_{J_i \in J(\mathcal{S},s,e)} w_i$.*

Proposition 15 allows us to decompose the computation of the weight of $\mathcal{S}$ according to the characteristics of each job. In the following, we use the notation $\delta(P)$ that equals 1 if proposition P holds, and 0 otherwise.

Proposition 15. *Let $\mathcal{S}$ be a feasible schedule for $k, s, e, \sigma, \varepsilon$, let $t \in [s, e)$, let τ' be the rpt-function of $\mathcal{S}$ at $t + 1$ and let V be the set of jobs that are processed at time t. $J(\mathcal{S}, s, e)$ is the direct sum of $J(\mathcal{S}, s, t)$, of $J(\mathcal{S}, t + 1, e)$ and of I:*

$$I = \{J_i | \ i \leq k, \ \ s \leq r_i < t + 1, \ \ \varepsilon(J_i) = 0, \ \ \tau'(J_i) + \delta(J_i \in V) > 0\} \tag{3.16}$$

Proof. We show that $J(\mathcal{S}, s, e) \subseteq J(\mathcal{S}, s, t) \cup J(\mathcal{S}, t + 1, e) \cup I$. Let $J_i \in J(\mathcal{S}, s, e)$. We have $i \leq k$, $s \leq r_i$ and $\varepsilon(J_i) = 0$ (otherwise less than m operations would be processed in $[s, e)$). If J_i is completed before t then $J_i \in J(\mathcal{S}, s, t)$. If J_i is such that $t + 1 \leq r_i$ then $J_i \in J(\mathcal{S}, t + 1, e)$. Hence, if J_i is completed after or at $t + 1$ and if $r_i < t$ then the rpt-function of $\mathcal{S}$ at t is strictly positive. Thus, $\tau'(J_i) + \delta(J_i \in V) > 0$ hence, $J_i \in I$.
It is obvious that $J(\mathcal{S}, s, t) \subseteq J(\mathcal{S}, s, e)$ and that $J(\mathcal{S}, t + 1, e) \subseteq J(\mathcal{S}, s, e)$. Now consider a job $J_i \in I$. Since $\tau'(J_i) + \delta(J_i \in V) > 0$, we know that at least one operation of J_i is processed. Hence, the whole job J_i is processed somewhere. Since $\varepsilon(J_i) = 0$ and since $s \leq r_i$, $J_i \in J(\mathcal{S}, s, e)$.
Finally, we show that the three sets do not intersect. First, it is obvious that $J(\mathcal{S}, s, t) \cap J(\mathcal{S}, t+1, e) = \emptyset$. Second, suppose that there exists a job $J_i \in J(\mathcal{S}, s, t) \cap I$. J_i is completed before or at t hence J_i is neither processed at t (*i.e.*, $\delta(J_i \in V) = 0$) nor after t (*i.e.*, $\tau'(J_i) = 0$); which contradicts $\tau'(J_i) + \delta(J_i \in V) > 0$. Third, suppose that $\exists J_i \in J(\mathcal{S}, t + 1, e)$ then $t + 1 \leq r_i$ and therefore the job does not belong to I. □

Recall that for each sub-problem we look for a feasible schedule whose weight in $[s, e)$ is maximum. We introduce some variables that represent this maximal weight.

Definition 11. *$W_k(s, e, \sigma, \varepsilon)$ is the weight in $[s, e)$ of the heaviest feasible schedules for $k, s, e, \sigma, \varepsilon$. If there is no such schedule, $W_k(s, e, \sigma, \varepsilon) = -\infty$.*

According to Proposition 14, there is an optimal well-ordered schedule. The rpt-function of this schedule after the completion of all jobs is a function ε_0 that equals 0 for each job. The rpt-function of this schedule before or at the minimal release date is defined over $\emptyset$. In the following, $\emptyset$ will denote this function. The maximum weighted number of early jobs is then $W_n(\min(\Theta), \max(\Theta), \emptyset, \varepsilon_0)$.

Informal Description

It now remains to show how the decomposition works. In this section, we describe it in an informal way. A formal presentation is provided in §3.3.4. Let now k be any value in $\{0, \ldots, n\}$ and let $[s, e]$ be any time interval. Let σ and ε be two functions and assume that there is a schedule $\mathcal{S}$ with weight $W_k(s, e, \sigma, \varepsilon)$.

- If J_k is not processed in $\mathcal{S}$ then, $W_k(s, e, \sigma, \varepsilon) = W_{k-1}(s, e, \sigma, \varepsilon)$.

- If at least one operation of J_k is processed in $[s, e]$, let t be the earliest time where J_k is processed after s, let τ and τ' be the rpt-functions at time t and $t + 1$ of $\mathcal{S}$, finally, let V be the set of jobs that are processed at time t (*cf.*, Figure 3.5). The basic idea is to decompose the problem into a left sub-problem (with $k - 1, s, t, \sigma, \tau$) and a right sub-problem (with $k, t + 1, e, \tau', \varepsilon$). There is a feasible schedule, as heavy as $\mathcal{S}$, that can be obtained as follows: Before t, follow a schedule that realizes $W_{k-1}(s, t, \sigma, \tau)$, at t process the jobs in V and finally, follow a schedule that realizes $W_k(t + 1, e, \tau', \varepsilon)$ after $t + 1$. The weight of this schedule is then equal to $W_{k-1}(s, t, \sigma, \tau) + W_k(t + 1, e, \tau', \varepsilon)$ plus the weights of "some" jobs that are neither counted in $W_{k-1}(s, t, \sigma, \tau)$ nor in $W_k(t + 1, e, \tau', \varepsilon)$ but that are processed in $[s, e)$. Actually these jobs are those of the set I as defined in Proposition 15.

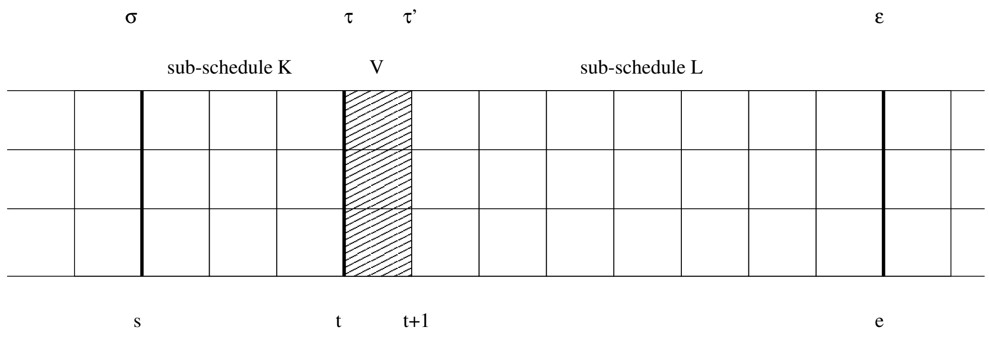


Figure 3.5: Decomposition scheme

This decomposition leads to a dynamic programming algorithm where we compute all the possible values of $W_k(s, e, \sigma, \varepsilon)$ are computed. Operations are processed at time points in Θ because feasible schedules are well-ordered. Hence, there are few time points to consider ($|\Theta| = O(n^2)$). However there are, *a priori*, $(m+1)^n$ different rpt-functions for each time point. Fortunately, well-ordered schedules are dominant and allow us to polynomially bound the number of relevant rpt-functions.

Decomposition

We define sets of functions that contain all rpt-functions at time t and $t+1$ of well-ordered schedules in which an operation of J_k is processed at t (*cf.*, Proposition 16).

Definition 12. *$\Psi_k^l(t)$ is the set of functions $\psi : \{J_i | i \leq k, r_i < t\} \to [0, m]$ such that (1) $\psi(J_i) > 0$ for at most l jobs, (2) $\psi(J_i)$ either equals 0 or is greater than or equal to $r_i + m - t$, and (3) $\psi(J_i) \leq \max(0, d_i - t)$.*

Proposition 16. *Let $\mathcal{S}$ be any feasible schedule for $k, s, e, \sigma, \varepsilon$ and let t be any time point at which an operation of J_k is processed. The rpt-functions of $\mathcal{S}$ at t and $t+1$ respectively belong to $\Psi_k^{m^2+1}(t)$ and to $\Psi_k^{m^2-m+1}(t+1)$.*

Proof. For any rpt-function τ at time t of a schedule, the conditions (2) and (3) of Definition 12 always hold: First, consider any job J_i such that $\tau(J_i) > 0$. Since $\tau(J_i) > 0$, the job is early and it cannot be completed before $r_i + m$. Hence, at most $r_i + m - t$ operations of J_i remain to be processed after t. Hence, $\tau(J_i) \geq r_i + m - t$. Second, notice that for any job J_i, at least $\max(0, d_i - t)$ operations can be processed after t, and hence, $\tau(J_i) \leq \max(0, d_i - (t+1))$.
Let τ and τ' be the rpt-functions of $\mathcal{S}$ at time t and $t+1$. $\mathcal{S}$ is well-ordered in $[s, e)$ and J_k is processed at t hence, at most $m^2 - m + 1$ jobs J_j, with $r_j \leq t$ and $j \leq k$, are completed strictly after time t. Thus, τ' takes a non-null value for less than $m^2 - m + 1$ jobs. Between t and $t+1$, at most m operations are processed. Therefore, τ takes a non-null value for less than $m^2 - m + 1 + m = m^2 + 1$ jobs. $\square$

Notice that there are at most $k^l m^l$ functions in each set $\Psi_k^l(t)$ (at most l jobs have to be picked among k and a value in $[1, m]$ has to be chosen for each of these jobs). The number of functions in the sets $\Psi_k^{m^2+1}(t)$ and $\Psi_k^{m^2-m+1}(t+1)$ is therefore polynomially bounded by n^{m^2+1}. This is very important because we will be able to enumerate all relevant rpt-functions at some time point in polynomial time.

From now on, k denotes any value in $\{0, \ldots, n\}$, $[s, e]$ is a time interval, σ and ε are two rpt-functions. We now introduce the notion of separators. A separator is a set of parameters V, τ' related to a time point $t \in [s, e)$. As shown in Proposition 17, it summarizes the "state" of the schedule at t: jobs in V are processed at t and the rpt-function of the schedule at $t+1$ is τ'.

Example. Consider the following well-ordered schedule of 8 jobs ($r_1 = 6$, $d_1 = 9$), ($r_2 = 5$, $d_2 = 8$), ($r_3 = 5$, $d_3 = 9$), ($r_4 = 5$, $d_4 = 9$), ($r_5 = 3$, $d_5 = 10$), ($r_6 = 1$, $d_6 = 10$), ($r_7 = 1$, $d_7 = 10$), ($r_8 = 2$, $d_8 = 15$). All the jobs are on-time in this schedule.

```
Time 1   2   3   4   5   6   7   8   9  10
M1   | 7 | 7 | 7 | . | 3 | 3 | 3 | 4 | 4 |
M2   | 6 | 6 | 6 | 8 | 4 | 1 | 1 | 1 | . |
M3   | . | 8 | 5 | 5 | 2 | 2 | 2 | 5 | 8 |
```

Let us illustrate the notion of separators on this example.

- If τ' denotes the rpt-function of the schedule at time 3 ($\tau'(J_6) = 1, \tau'(J_7) = 1, \tau'(J_8) = 2$) then $(2, \{6, 7, 8\}, \tau')$ is a separator for $k = 9, s = 1, e = 17, \sigma = \emptyset, \varepsilon = \varepsilon_0$. The rpt-functions $\emptyset$ and ε_0, as defined in the end of Section 3.3.4, are the rpt-functions of the schedule at time $\min(\Theta) = 1$ and $\max(\Theta) = 17$. In the following, we will see that the schedule can be decomposed into two sub-schedules, "before" and "after" the separator $(2, \{6, 7, 8\}, \tau')$.

- If τ'' denotes the rpt-function of the schedule at time 5 ($\tau''(J_5) = 1, \tau''(J_6) = 0, \tau''(J_7) = 0, \tau''(J_8) = 1$) then $(4, \{8, 5\}, \tau'')$ is a separator for $k = 9, s = 3, e = 17, \sigma = \tau', \varepsilon = \varepsilon_0$.

Definition 13. *The triplet (t, V, τ') is a separator for $k, s, e, \sigma, \varepsilon$ if and only if*

1. *t and $t + 1$ are time points of Θ and $t \in [\max(s, r_k), \min(e, d_k))$,*

2. *$V \ni J_k$ is a subset of at most m jobs in $\{J_i | i \leq k, r_i \leq t < d_i\}$,*

3. *if $s \leq r_k$, $\tau'(J_k) = m - 1$ otherwise, $\tau'(J_k) = \sigma(J_k) - 1$,*

4. *$\forall i \leq k$, if $r_i < s$ then $\sigma(J_i) \geq \tau'(J_i) + \delta(J_i \in V)$; if $r_i < t + 1$, $\tau'(J_i) \geq \varepsilon(J_i)$.*

Proposition 17. *Let $\mathcal{S}$ be any feasible schedule for $k, s, e, \sigma, \varepsilon$ and assume that $t \in [s, e)$ is the first time-point at which an operation of J_k is processed. Let V be the set of jobs that are processed at time t and let τ' be the rpt-function of $\mathcal{S}$ at $t + 1$. The triplet (t, V, τ') is a separator for $k, s, e, \sigma, \varepsilon$.*

Proof. We first prove that the first and the second conditions of Definition 13 hold. t and $t + 1$ are time points of Θ because $\mathcal{S}$ is well-ordered in $[s, e)$. On top of that, $t \in [\max(s, r_k), \min(e, d_k))$ because operations of J_k are processed in $[r_k, d_k)$. J_k obviously belongs to V and at most m jobs are processed at t. The jobs J_i that are processed at t are such that $i \leq k$ and $r_i \leq t < d_i$ (recall that $\mathcal{S}$ is a feasible schedule for $k, s, e, \sigma, \varepsilon$).
We now prove that the third condition of Definition 13 holds. If $r_k \geq s$, then no operation of J_k is processed before s and since no operation of J_k is processed in $[s, t)$ (*cf.*, definition of t), J_k is not processed before t. On top of that, J_k is early because it is processed at t hence, $m - 1$ operations remain to be processed after $t + 1$, *i.e.*, $\tau'(J_k) = m - 1$. Now assume that $r_k < s$, then because no operation of J_k is processed in $[s, t)$, it follows that the number of operations that remain to be processed at s and at t are the same, and hence $\sigma(J_k) = \tau'(J_k) + 1$.
We now prove that the last condition of Definition 13 holds. It holds because the functions $\sigma(J_i), \tau'(J_i) + \delta(J_i \in V), \tau'(J_i)$ and $\varepsilon(J_i)$ are the rpt-functions of the same schedule at successive time points $s \leq t < t + 1 \leq e$. $\square$

Proposition 23 introduces a simple relation between the variables W. We need several technical propositions (Propositions 18, 19, 20, 22, 21) to achieve the proof. In the following, we define

$\tau(J_i) = \tau'(J_i) + \delta(J_i \in V)$, and we denote W' the maximum of (3.17) over all separators (t, V, τ') of $k, s, e, \sigma, \varepsilon$ under the constraint $\tau' \in \Psi_k^{m^2-m+1}(t+1)$ (if no such separator exists, $W' = -\infty$).

$$W_{k-1}(s,t,\sigma,\tau) + W_k(t+1,e,\tau',\varepsilon) + \sum_{\substack{i \leq k \\ s \leq r_i \leq t \\ \varepsilon(J_i) = 0 \\ \tau'(J_i) + \delta(J_i \in V) > 0}} w_i \tag{3.17}$$

Proposition 18. *If W' takes a finite value, then there is a feasible schedule for $k, s, e, \sigma, \varepsilon$ in which J_k is processed between s and e.*

Proof. Let (t, V, τ') be the separator that realizes W'. $W_{k-1}(s,t,\sigma,\tau)$ and $W_k(t+1,e,\tau',\varepsilon)$ also take finite values; hence, there are two schedules $\mathcal{K}$ and $\mathcal{L}$ that realize, respectively, $W_{k-1}(s,t,\sigma,\tau)$ and $W_k(t+1,e,\tau',\varepsilon)$. We build a schedule $\mathcal{S}$ as follows: (1) strictly before time t, follow $\mathcal{K}$, (2) at time t, process jobs in V, (3) strictly after t, follow the schedule $\mathcal{L}$. This schedule has to be completed because some operations of J_k may have to execute before s due to the value of $\sigma(J_k)$. So we have a fourth step: (4) if $r_k < s$, schedule consecutively $m-\sigma(J_k)$ operations of J_k during $[s-(m-\sigma(J_k)), s)$ (recall that there is no machine constraint before s). In this last step, we claim that operations are not processed before their release dates. Indeed, σ is a rpt-function thus, because $\sigma(J_k) > 0$, we have $\sigma(J_k) \geq r_k + m - s$.
We claim that $\mathcal{S}$ is feasible for k, s, e, σ. It is obvious that no more than m operations are processed at the same time in $[s, e)$ and that jobs $J_{k+1}, \ldots, J_n$ are not processed. The rpt-function of $\mathcal{S}$ at time e is the one of $\mathcal{L}$ and hence, it is equal to ε. Similarly, the restrictions to the $k-1$ first jobs of the rpt-function of $\mathcal{S}$ at time s and of σ, are equal. We have to check that these functions are equal for J_k (if they are defined over these jobs). J_k is early and by construction of $\mathcal{S}$, $m - \sigma(J_k)$ operations of J_k are processed before s hence, the remaining processing time of J_k at s is exactly $\sigma(J_k)$.
Finally, we have to show that $\mathcal{S}$ is "well-ordered" over $[s, e)$. Given the definition of separators, and since the sub-schedules $\mathcal{K}$ and $\mathcal{L}$ are well-ordered, operations start and are completed at time points in Θ. We now examine the second condition of the definition of well-ordered schedules. Let J_i be any job and let x be any time point in $[s, e)$ at which one operation of J_i is processed. If $t < x$, then any job J_j that is processed strictly after time x in $\mathcal{S}$ is processed at the same time in $\mathcal{L}$. Hence, the constraint holds for $\mathcal{S}$ because it holds for $\mathcal{L}$. If $x < t$, then remark that (1) any job J_j (with $j \neq k$) which is processed strictly after x and strictly before t is processed at the same time in $\mathcal{K}$ and (2) that any job J_j with $r_j \leq x$ which is processed after t is such that $\tau(J_j) > 0$; hence, it is also processed after t in $\mathcal{K}$. Consequently, the constraint holds for $\mathcal{S}$ because it holds for $\mathcal{K}$ and because no operation of J_k is processed in $[s, x)$. The case $x = t$ remains. We prove that the constraint holds for $J_i = J_k$ (it will consequently hold for all values of $i \leq k$): Since $\tau' \in \Psi_k^{m^2-m+1}(t+1)$, at most $m^2 - m + 1$ of the jobs J_j with $j \leq k, r_j < t+1$ are such that $\tau'(J_j) > 0$. Thus at most $m^2 - m + 1$ of the same jobs are processed after t. $\square$

Proposition 19. *Let feasible schedules for $k, s, e, \sigma, \varepsilon$, where J_k is processed in $[s, e)$, exist. Then W' is the weight of the heaviest schedule among them.*

Proof. Let $\mathcal{S}$ be the heaviest schedule in which some operations of J_k are processed in $[s, e)$ (such a schedule exists because of our hypothesis). Let then t, V, τ, τ' be, respectively, the first time-point at which J_k is processed, the set of jobs that are processed at t and the rpt-functions of $\mathcal{S}$ at t and

$t+1$. Let us now compute the weight $\sum_{J(\mathcal{S},s,e)} w_i$ of $\mathcal{S}$. According to Proposition 15 it equals

$$\begin{aligned} & \sum_{J(\mathcal{S},s,t)} w_i + \sum_{J(\mathcal{S},t+1,e)} w_i + \sum_{\substack{i \le k \\ s \le r_i \le t \\ \varepsilon(J_i)=0 \\ \tau'(J_i)+\delta(J_i \in V) > 0}} w_i \\ \le \quad & W_{k-1}(s,t,\sigma,\tau) + W_k(t+1,e,\tau',\varepsilon) + \sum_{\substack{i \le k \\ s \le r_i \le t \\ \varepsilon(J_i)=0 \\ \tau'(J_i)+\delta(J_i \in V) > 0}} w_i \end{aligned} \tag{3.18}$$

Notice that (t,V,τ') is a separator (Proposition 17) and that τ and τ' belong to $\Psi_k^{m^2+1}(t)$ and of $\Psi_k^{m^2-m+1}(t+1)$ (Proposition 16). Thus, according to the definition of W', the above equation leads to:

$$\sum_{J_i \in J(\mathcal{S},s,e)} w_i \le W' \tag{3.19}$$

□

Proposition 20. *If $W_{k-1}(s,e,\sigma,\varepsilon)$ takes a finite value and if either (1) $r_k < s$ and $\sigma(J_k) = \varepsilon(J_k)$ or (2) $s \le r_k < e$ and $\varepsilon(J_k) = 0$ or (3) $s \le r_k < e$ and $\varepsilon(J_k) = m$ or (4) $e \le r_k$ then there is a feasible schedule for k,s,e,σ where J_k is not processed in $[s,e)$.*

Proof. Let $\mathcal{S}$ be a schedule that realizes $W_{k-1}(s,e,\sigma,\varepsilon)$. We modify the schedule $\mathcal{S}$ by adding eventually some operations before s and after e. Since an unlimited number of operations can be processed in parallel before s and after e, such additions are valid. The proof follows the cases introduced in Proposition 20.

1. $r_k < s$ and $\sigma(J_k) = \varepsilon(J_k)$. First notice that if $\sigma(J_k) = 0$ then $\mathcal{S}$ is feasible for k,s,e,σ,ε. If $\sigma(J_k) > 0$ then, because σ is a rpt-function, we know that $\sigma(J_k) \ge r_k + m - s$. Hence, we can schedule $m - \sigma(J_k)$ operations before s. Similarly, we can schedule $\varepsilon(J_k)$ operations after e. The overall schedule is then feasible for k,s,e,σ,ε.

2. $s \le r_k < e$ and $\varepsilon(J_k) = 0$. $\mathcal{S}$ is feasible for k,s,e,σ,ε (J_k is late in this schedule)

3. $s \le r_k < e$ and $\varepsilon(J_k) = m$. m operations are added after e (this is possible because $\varepsilon(J_k) \le \max(0, d_k - t)$ leads to $m \le d_k - t$). The overall schedule is then feasible for k,s,e,σ,ε.

4. $e \le r_k$. $\mathcal{S}$ is feasible for k,s,e,σ,ε.

□

Proposition 21. *Let feasible schedules for k,s,e,σ,ε where J_k is not processed in $[s,e)$ exist. Then either (1) $r_k < s$ and $\sigma(J_k) = \varepsilon(J_k)$ or (2) $s \le r_k < e$ and $\varepsilon(J_k) = 0$ or (3) $s \le r_k < e$ and $\varepsilon(J_k) = m$ or (4) $e \le r_k$.*

Proof. We distinguish three cases depending on the relative positions of r_k and $[s,e)$.

- $r_k < s$. The remaining processing times of J_k at s and e are, respectively, $\sigma(J_k)$ and $\varepsilon(J_k)$. Therefore, $\sigma(J_k) - \varepsilon(J_k)$ operations of J_k are processed in $[s,e)$. Given our hypothesis, this leads to $\sigma(J_k) = \varepsilon(J_k)$.

- $s \le r_k < e$. Either J_k is not processed in the schedule (it is late) and thus $\varepsilon(J_k) = 0$ or it is fully processed after e and thus $\varepsilon(J_k) = m$.

- $e \le r_k$.

All together, this leads to the four conditions of the proposition. □

Proposition 22. *Let feasible schedules for k, s, e, σ where J_k is not processed in $[s, e)$ exist. Then $W_{k-1}(s, e, \sigma, \varepsilon)$ is the weight of the heaviest among them.*

Proof. Let $\mathcal{S}$ be any feasible schedule for k, s, e, σ in which J_k is not processed in $[s, e)$ and whose weight $\sum_{J_i \in J(\mathcal{S},s,e)} w_i$ is maximal. The weight of J_k is not taken into account (*cf.*, definition of $J(\mathcal{S}, s, e)$). Hence, $\sum_{J_i \in J(\mathcal{S},s,e)} w_i$ equals the weight of the same schedule in which J_k has been removed. This weight is lower than or equal to $W_{k-1}(s, e, \sigma, \varepsilon)$. Hence, there exists a schedule that realizes $W_{k-1}(s, e, \sigma, \varepsilon)$. We could add some operations of J_k before s and after e to transform this schedule into a feasible schedule for $k, s, e, \sigma, \varepsilon$ (similar as in the proof of Proposition 20). Therefore, we could prove that $W_{k-1}(s, e, \sigma, \varepsilon)$ is the weight of a schedule in which J_k is not processed in $[s, e)$. □

Proposition 23. *If either (1) $r_k < s$ and $\sigma(J_k) = \varepsilon(J_k)$ or (2) $s \leq r_k < e$ and $\varepsilon(J_k) = 0$ or (3) $s \leq r_k < e$ and $\varepsilon(J_k) = m$ or (4) $e \leq r_k$, then $W_k(s, e, \sigma, \varepsilon) = \max(W', W_{k-1}(s, e, \sigma, \varepsilon))$. If none of the four conditions holds, $W_k(s, e, \sigma, \varepsilon)$ equals W'.*

Proof. We first prove that, $W_k(s, e, \sigma, \varepsilon) \geq W'$. This is obvious if W' does not take a finite value. Now assume that it does. Hence, there is a feasible schedule for k, s, e, σ in which J_k is processed in $[s, e)$ (Proposition 18) and W' is the weight of the heaviest of these schedules (Proposition 19). Thus, $W_k(s, e, \sigma, \varepsilon) \geq W'$.
We now prove that, if one of the four conditions holds then $W_k(s, e, \sigma, \varepsilon) \geq W_{k-1}(s, e, \sigma, \varepsilon)$. According to Proposition 20, there is a feasible schedule for k, s, e, σ in which J_k is not processed in $[s, e)$. Note that $W_{k-1}(s, e, \sigma, \varepsilon)$ is the weight of the heaviest of these schedules. Hence, $W_k(s, e, \sigma, \varepsilon) \geq W_{k-1}(s, e, \sigma, \varepsilon)$.
We now prove that $W_k(s, e, \sigma, \varepsilon) \leq \max(W', W_{k-1}(s, e, \sigma, \varepsilon))$. Consider a schedule that realizes $W_k(s, e, \sigma, \varepsilon)$. Either J_k is processed in $[s, e)$ and thus $W_k(s, e, \sigma, \varepsilon) = W'$ or it is not and thus $W_k(s, e, \sigma, \varepsilon) = W_{k-1}(s, e, \sigma, \varepsilon)$ (*cf.*, Proposition 19 and 22). Hence, $W_k(s, e, \sigma, \varepsilon)$ is lower than or equal to $\max(W', W_{k-1}(s, e, \sigma, \varepsilon))$.
Finally, notice that if none of the four conditions holds then J_k is processed in $[s, e)$ in any feasible schedule for k, s, e, σ (Proposition 21). Thus, $W_k(s, e, \sigma, \varepsilon) = W'$. □

3.3.5 A Dynamic Programming Algorithm

As shown in § 3.3.4, the maximum weighted number of early jobs is exactly $W_n(\min(\Theta), \max(\Theta), \emptyset, \varepsilon_0)$. It can be dynamically computed thanks to the decomposition scheme (Theorem 23). Algorithm 3 is a straight implementation of this scheme: Given a combination of the parameters $k, s, e, \sigma, \varepsilon$, the value of $W_k(s, e, \sigma, \varepsilon)$ is recursively computed (lines 6–27). Note that Algorithm 3 relies on Definition 9 for the *initialization part*: $W_0(s, e, \sigma, \varepsilon) = 0$. This is omitted in the pseudo-code. To simplify the presentation, we use σ and ε in the expressions $W_{k-1}(s, t, \sigma, \tau)$ (line 12) and $W_{k-1}(s, e, \sigma, \varepsilon)$ (line 26) while it would be more appropriate to refer to the restriction of σ and ε to the $k - 1$ first jobs. Note that the algorithm does not try to reach a value of W (lines 12, 26) that has not been previously computed.

We do not have to consider all possible combinations of the parameters (lines 1–6) and we only focus on "*non-dominated*" combinations $k, s, e, \sigma, \varepsilon$, *i.e.*, combinations where $s, e \in \Theta$, $\sigma \in \Psi_k^{m^2-m+1}(s)$ and $\varepsilon \in \Psi_k^{m^2+1}(e)$. Note that according to Definition 12, $\Psi_n^{m^2-m+1}(\min(\theta))$ contains the function "$\emptyset$" so, $n, \min(\theta), \max(\theta), \emptyset, \varepsilon_0$ is a non-dominated combination and thus the algorithm computes $W_n(\min(\Theta), \max(\Theta), \emptyset, \varepsilon_0)$. It is *correct* to restrict the search to non-dominated combinations because, in the recursion formula of Theorem 23, all combinations that are examined are non-dominated: Given the definition of W' (Equation 3.17), it is easy to check that $k - 1, s, t, \sigma, \tau$ and $k, t + 1, e, \tau', \varepsilon$ are non-dominated combinations.

Each function $\psi \in \Psi_k^l(t)$ can be encoded as an array of size n that provides the value taken by the function over each job. It can also be encoded as an array of size $2l$ that provides the indices of the jobs for which $\psi(J_i) > 0$ and the corresponding values $\psi(J_i)$ (this encoding relies on the fact that less than l jobs are such that $\psi(J_i) > 0$). We will rely on the second implementation because the relevant values of l ($l = m^2 - m + 1$ and $l = m^2 + 1$) do not depend of n and therefore the overall encoding of a function requires a constant amount of space. Recall that in each set there are $O(n^{m^2+1})$ functions. Since $|\Theta| = O(n^2)$, the space required to store all functions is $O(n^{m^2+3})$. We will also store the values $W_k(s, e, \sigma, \varepsilon)$ in a multi-dimensional array. Actually, for each value of s and e, the functions of $\Psi_k^{m^2-m+1}(s)$ and of $\Psi_k^{m^2+1}(e)$ can be indexed (there are, respectively, $O(n^{m^2-m+1})$ and $O(n^{m^2+1})$ such functions). The index can then be used to store $W_k(s, e, \sigma, \varepsilon)$ in an array $W[k][i_s][i_e][i_\sigma][i_\varepsilon]$:

- where i_s is the index of s in Θ,
- where i_e is the index of e in Θ,
- where i_σ is the index of σ in $\Psi_k^{m^2-m+1}(s)$,
- where i_ε is the index of ε in $\Psi_k^{m^2+1}(e)$.

The size of the array W is therefore $O(nn^2n^2n^{m^2+1}n^{m^2-m+1}) = O(n^{2m^2-m+7})$.

It is now easy to compute the time-complexity of the algorithm:

- there are n possible values for k (line 1),
- there are $O(n^2)$ possible values for s (line 2),
- there are $O(n^2)$ possible values for e (line 3),
- there are $O(n^{m^2-m+1})$ functions σ (line 4),
- there are $O(n^{m^2+1})$ functions ε (line 5),
- there are $O(n^2)$ possible values for t distinct values (line 7),
- there are $O(n^{m^2-m+1})$ functions τ' (line 8),
- there are at most n^{m-1} subsets V (line 9),
- finding the function τ (*i.e.*, the index of the function) may require to browse the whole set $\Psi_k^{m^2+1}(t)$; which may requires $O(n^{m^2+1})$ steps (line 11).

This leads to an overall complexity of $O(n^{4m^2-m+10})$. Note that the algorithm is still valid when weights take non-integer values.

3.3.6 Open Problems

For any fixed value of m, the parallel machine problem can be solved in $O(n^{4m^2-m+10})$. The optimal solution is transformed into the optimum solution of the open-shop problem in $O(mn\log^2(mn))$ (*cf.*, § 3.3.2). Hence, the problem $Om|p_{ij} = 1, r_i|\sum w_iU_i$ is solvable in polynomial time. The very high time and space complexity of the algorithm that we have presented, make it hardly usable in practice. We think that the complexity could be slightly decreased by using some additional data structures. However, this would not make the algorithm more usable in practice and it would make the algorithm much more complex.

An interesting open question concerns the status of the same UET open-shop problems with other criteria, such as the weighted flow time or the weighted tardiness. A straight generalization of our algorithm is not possible because the dominance property of Proposition 14, based upon an exchange argument, would not hold.

Algorithm 3 Computation of the values $W_k(s, e, \sigma, \varepsilon)$

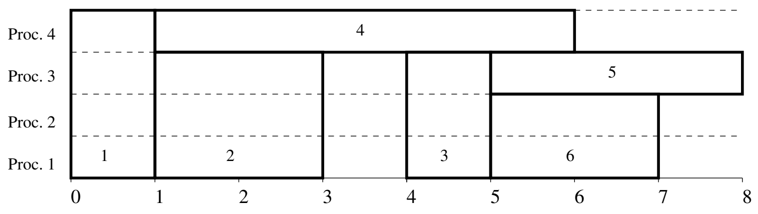
```
for $k = 1$ to $n$ do
  for $s \in \Theta$ (values taken in decreasing order) do
    for $e \in \Theta, e \geq s$ (values taken in increasing order) do
      for $\sigma \in \Psi_k^{m^2-m+1}(s)$ do
        for $\varepsilon \in \Psi_k^{m^2+1}(e)$ do
          $W_k(s, e, \sigma, \varepsilon) \leftarrow -\infty$
          for $t \in \Theta \cap [s, e)$ do
            for $\tau' \in \Psi_k^{m^2-m+1}(t+1)$ do
              for all $V$ subset of at most $m$ jobs including $J_k$ do
                if $(t, V, \tau')$ is a separator for $k, s, e, \sigma, \varepsilon$ then
                  Find in $\Psi_k^{m^2+1}(t)$ the fct. $\tau$ s.t. $\forall J_i | r_i < t, \tau(J_i) = \tau'(J_i) + \delta(J_i \in V)$
                  $S \leftarrow W_{k-1}(s, t, \sigma, \tau) + W_k(t+1, e, \tau', \varepsilon)$
                  for $i = 1$ to $k$ do
                    if $J_i$ belongs to $I$ as defined in Lemma 15 then
                      $S \leftarrow S + w_i$
                    end if
                  end for
                  if $W_k(s, e, \sigma, \varepsilon) < S$ then
                    $W_k(s, e, \sigma, \varepsilon) \leftarrow S$
                  end if
                end if
              end for
            end for
          end for
          if (1) $r_k < s$ and $\sigma(J_k) = \varepsilon(J_k)$ or (2) $s \leq r_k < e$ and $\varepsilon(J_k) = 0$ or (3) $s \leq r_k < e$
          and $\varepsilon(J_k) = m$ or (4) $e \leq r_k$ then
            $W_k(s, e, \sigma, \varepsilon) \leftarrow \max(W_k(s, e, \sigma, \varepsilon), W_{k-1}(s, e, \sigma, \varepsilon))$
          end if
        end for
      end for
    end for
  end for
end for
```

Chapter 4

Scheduling Equal Length Multiprocessor Tasks

We study the situation where a set of n tasks has to be scheduled on m parallel processors. In most of the classical scheduling models, it is assumed that each task is processed on one processor (also called a machine) at a time. However, the relatively recent development of multiprocessor computer systems and of complex manufacturing environments has led researchers to study more complex situations where each task requires simultaneously several processors (see for instance [35, 88]).

In this chapter we consider a particular multiprocessor scheduling environment where, for each task i, a fixed number $size_i$ of processors is required to execute the task. Yet, the processors required are not specified. Following the classical scheduling notation (see for instance [35, 38]), the corresponding scheduling problems are referred to as $P|size_i|F$. See Figure 4.1 for an instance of $P|r_i, size_i|C_{\max}$.



task	size	p	r
1	4	1	0
2	3	2	0
3	3	1	4
4	1	5	0
5	1	3	4
6	2	2	4

Figure 4.1: An Optimal Schedule of a $P4|size_i|C_{\max}$ Instance

When tasks have identical processing times, the multiprocessor problem $Pm|size_i, p_i = p|F$ is solvable in polynomial time for $F \in \{\sum w_iC_i, \sum w_iU_i, \sum T_i\}$ (see [48] for a review). When the number of processors is not fixed (*i.e.*, m is a data), the $C_{\max}$ and $\sum C_i$ problems are binary NP-Hard ([160] and [89]). So, $Pm|size_i, p_i = p|\sum w_iT_i$ is the only remaining open question.

With identical processing times and arbitrary release dates, the situation is more complex. Of course, $Pm|r_i, size_i, p_i = p|C_{\max}$, as the counterpart of $Pm|size_i, p_i = p|L_{\max}$, is solvable in polynomial time. Moreover, a linear time algorithm is proposed in [48] for $P2|r_i, size_i, p_i = p|\sum C_i$ and it is conjectured that the 3-machine problem is polynomial.

In this Chapter, we first show (Section 4.1) that Brucker, Knust, Roper and Zinder's conjecture holds for any number of machines: $Pm|r_i, size_i, p_i = p|\sum C_i$ can be solved in polynomial time by dynamic programming. Then we study the unit execution time problem ($p_i = 1$) and we show that the minimum maximal tardiness can be computed in polynomial time when $size_i \in \{1, m\}$ (Section 4.2). In this problem, there are no more than two possible sizes, either 1 (small tasks) or m (tall tasks), and we will refer to it as the "tall/small" problem.

4.1 Minimizing Total Completion Time, $Pm|r_i, p_i = p, size_i|\sum C_i$

We introduce an algorithm for $Pm|r_i, p_i = p, size_i|\sum C_i$ and $Pm|r_i, p_i = p, size_i|C_{\max}$. A dominance property is introduced in Section 4.1.1. In Section 4.1.2, we define the variables of the dynamic programming algorithm. The algorithm itself is described in Section 4.1.3.

4.1.1 Dominance Property

From now on let F be the objective function (either $C_{\max}$ or $\sum C_i$) and assume that tasks are sorted in non-decreasing order of release dates, *i.e.*, $r_1 \leq r_2 \ldots \leq r_n$. Proposition 24 is a strong dominance property for optimal schedules.

Proposition 24. *There is an optimal schedule in which, for any pair of tasks i, j with identical size, $i < j \Rightarrow C_i \leq C_j$.*

Proof. Consider an optimal solution for F that lexicographically minimizes the vector of completion times $(C_1, \ldots, C_n)$. Assume that there are two tasks i, j with $size_i = size_j$, $i < j$ and $C_i > C_j$. Let us exchange i and j. Since $i < j \Rightarrow r_i \leq r_j$, the exchange does not violate the release dates. It is easy to see that this exchange does not modify the objective function F (because $F = C_{\max}$ or $F = \sum C_i$). So we have an optimal schedule with a lexicographically smaller vector of completion times. This contradicts our hypothesis. □

4.1.2 Variables Definition

We use dynamic programming to compute the optimal schedule. Each sub-problem is defined by two tuples of m variables: processors variables $(t_1, \ldots, t_m)$ and tasks variables $(i_1, \ldots, i_m)$.

- **Processors variables** $(t_1, \ldots, t_m)$ represent the availability of the processors: No processor is available before t_1, 1 processor is available between t_1 and t_2, 2 processors are available between t_2 and t_3, $\ldots$, m processors are available after t_m (see Figure 4.2).
- **Tasks variables** $(i_1, \ldots, i_m)$ define the subset of the tasks that are considered in the sub-problem (the other ones have already been scheduled): For a given size s, we only consider tasks of this size with $i \geq i_s$. In other words, the tasks of the sub-problem are exactly

$$\bigcup_{s=1}^{m} \{i | i \geq i_s, size_i = s\}.$$

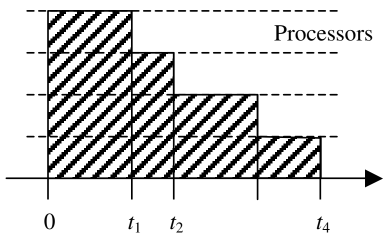


Figure 4.2: Processors Variables

To simplify the presentation of the algorithm, we **assume** that the processors variables are such that $t_1 \leq t_2 \leq \ldots \leq t_m$ and $t_m - t_1 \leq p$. In the following we will see that this assumption always holds at any step of the algorithm.

Processors variables together with tasks variables define a sub-problem. Five sub-problems are depicted in Figure 4.3. For each of them we have a partial schedule made of tasks that have already been scheduled. The tasks of the sub-problem are those that are not scheduled yet. They have to be scheduled after the "thick border line" defined by the processors variables.

In the following, $F(t_1, \ldots, t_m, i_1, \ldots, i_m)$ denotes the optimal value of the objective function for the corresponding sub-problem. Note that the optimum of our initial problem is exactly

$$F(\min_i r_i, \ldots, \min_i r_i, 0, \ldots, 0).$$

4.1.3 Dynamic Programming

Consider an optimal non-dominated active schedule of the sub-problem and assume that s is the size of one of the tasks of the sub-problem with minimal completion time (in the schedule of Figure 4.1, $s = 4$). Since the tasks of size s are exactly $\{i | i \geq i_s, size_i = s\}$ and since completion times of the tasks of the same size are non-decreasing (Proposition 24), we know that the task $x = \min\{i | i \geq i_s, size_i = s\}$ is a task with minimum completion time.

We distinguish two cases to recursively compute $F(t_1, \ldots, t_m, i_1, \ldots, i_m)$. First, if $t_1 < r_x$ then at least one processor is available before r_x. Since no task starts before r_x (otherwise the completion time of x would not be minimal), the processors variables can be "increased" so that no processor is available before r_x. Second, if $t_1 \geq r_x$ then x starts exactly at time t_s and we will see how tasks and processors variables can be computed.

CASE 1: $t_1 < r_x$.

Since no task starts before r_x (otherwise the completion time of x would not be minimal), the objective function $F(t_1, \ldots, t_m, i_1, \ldots, i_m)$ equals

$$F(\max(t_1, r_x), \ldots, \max(t_m, r_x), i_1, \ldots, i_m). \tag{4.1}$$

Note that we have $\max(t_1, r_x) \leq \max(t_2, r_x) \leq \ldots \leq \max(t_m, r_x)$ and that $\max(t_m, r_x) - \max(t_1, r_x) \leq p$. Hence, the two initial assumptions on the processors variables still hold.

CASE 2: $r_x \leq t_1$.

Task x starts exactly at time t_s and is completed at $t_s + p$. The new tasks variables are $(i_1, \ldots, i_{s-1}, x + 1, i_{s+1}, \ldots, i_m)$. Let us compute the new processors variables. Task x is completed at $t_s + p$ and the s processors that were used to execute the tasks are available again. The s processors that used to be available at t_s are now available at $t_s + p$ and nothing changes for the $m - s$ other processors. Since $t_m - t_1 \leq p$, the new processors variables are exactly

$$(\underbrace{t_{s+1}, \ldots, t_m}_{m-s \text{ elements}}, \underbrace{t_s + p, \ldots, t_s + p}_{s \text{ elements}}).$$

It is easy to see that $(t_s + p) - t_{s+1} \leq p$ and consequently, the difference between the last and the first processor variable is still lower than or equal to p. Hence, the two initial assumptions on the processors variables still hold.

For the $C_{\max}$ criterion, $F(t_1, \ldots, t_m, i_1, \ldots, i_m)$ equals

$$F(t_{s+1}, \ldots, t_m, t_s + p, \ldots, t_s + p, i_1, \ldots, i_{s-1}, x + 1, i_{s+1}, \ldots, i_m). \tag{4.2}$$

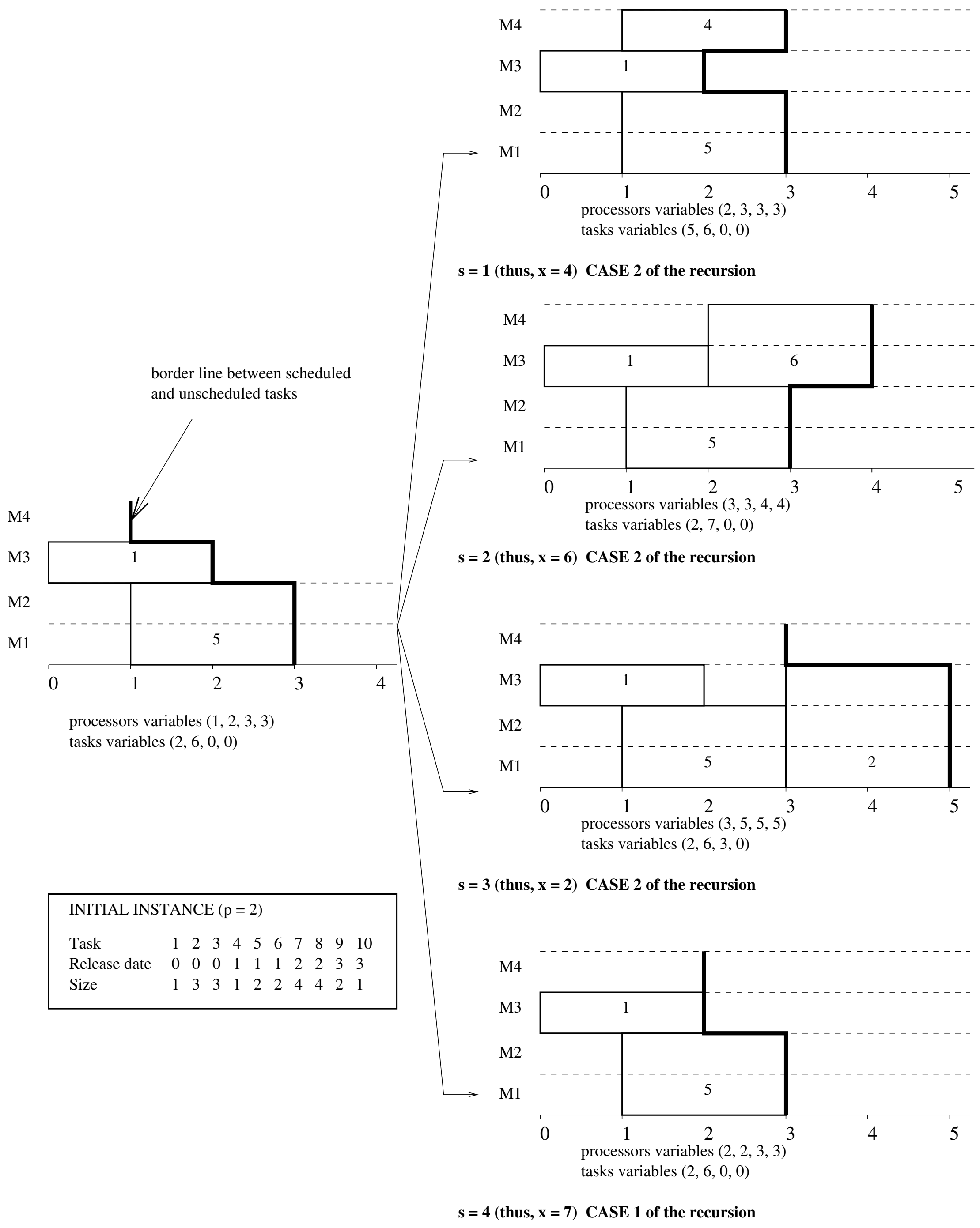
For the $\sum C_i$ criterion, $F(t_1, \ldots, t_m, i_1, \ldots, i_m)$ equals

$$F(t_{s+1}, \ldots, t_m, t_s + p, \ldots, t_s + p, i_1, \ldots, i_{s-1}, x + 1, i_{s+1}, \ldots, i_m) + t_s + p. \tag{4.3}$$

So, if we know the size s of the task of the sub-problem with minimal completion time, then, thanks to (4.1), (4.2) and (4.3), we can recursively compute the optimum value of $F(t_1, \ldots, t_m, i_1, \ldots, i_m)$. Since s is not known in advance, all possible values are tried and the one leading to the best solution is chosen. On the example of Figure 4.3, we can see that the left sub-problem leads to 4 sub-problems (one for each possible size). The last one only corresponds to CASE 1.

Note that the variables t_i can take any integer value. Hence, there is not a polynomial number of states and thus our dynamic programming scheme is not likely to run in polynomial time. However, it's easy to see that on an active schedule, start times equal a release date modulo p. More precisely, tasks start and are completed in

$$\Theta = \{t \mid \exists r_i, \exists l \in \{0, \ldots, n\}, t = r_i + lp\}.$$


border line between scheduled and unscheduled tasks
processors variables (1, 2, 3, 3)
tasks variables (2, 6, 0, 0)
processors variables (2, 3, 3, 3)
tasks variables (5, 6, 0, 0)
s = 1 (thus, x = 4) CASE 2 of the recursion
processors variables (3, 3, 4, 4)
tasks variables (2, 7, 0, 0)
s = 2 (thus, x = 6) CASE 2 of the recursion
processors variables (3, 5, 5, 5)
tasks variables (2, 6, 3, 0)
s = 3 (thus, x = 2) CASE 2 of the recursion
processors variables (2, 2, 3, 3)
tasks variables (2, 6, 0, 0)
s = 4 (thus, x = 7) CASE 1 of the recursion
M1
M2
M3
M4

INITIAL INSTANCE (p = 2)

Task	1	2	3	4	5	6	7	8	9	10
Release date	0	0	0	1	1	1	2	2	3	3
Size	1	3	3	1	2	2	4	4	2	1

Figure 4.3: The Dynamic Scheme

Since there is an optimal active schedule and since there are $O(n^2)$ time points in Θ, we have no more than $O(n^{2m})$ relevant tuples of processors variables to consider.

Algorithm 4 is a straight implementation of the decomposition rules for $C_{\max}$ (for $\sum C_i$, lines 6 and 16 have to be changed respectively into $F(t_1, \ldots, i_1, \ldots, i_m) = 0$ and $F_s^* \to F(t_{s+1}, \ldots, t_m, t_s + p, \ldots, t_s + p, i_1, \ldots, i_{s-1}, x + 1, i_{s+1}, \ldots, i_m) + t_s + p)$. Before describing Algorithm 4, we need to introduce the notation $N(u, z)$, the smallest index greater than u among tasks of size z (*i.e.*, $N(u, z) = \min\{i | i \geq u, \mathit{size}_i = z\}$). If such a task does not exist, $N(u, z)$ is set to a special negative value. Note that $N(u, z)$ can be pre-computed in linear time. Now, let us examine Algorithm 4.

- We iterate over all tasks and processors variables sorted in decreasing lexicographical order (lines 1–4). This ordering ensures that the optima of all sub-problems of the current sub-problem have already been computed.

- If the set of tasks of the sub-problem $\bigcup_{s=1}^{m} \{i | i \geq i_s, \mathit{size}_i = s\}$ is empty (line 5) then $F(t_1, \ldots, i_1, \ldots, i_m)$ equals t_m.

- If the set of tasks of the sub-problem is not empty, we iterate over all possible sizes s of the tasks of the sub-problem (lines 9–10). For each s, F_s^*, the optimum of the sub-problem, under the hypothesis that first task to execute has size s, is computed (lines 12–13 and 14–20 respectively correspond to CASE 1 and CASE 2).

- If $t_s + p \notin \Theta$ (line 18) then we are building a non-active schedule so, F_s^* can be set to ∞.

- If $t_s + p \in \Theta$ (line 15) then $(t_{s+1}, \ldots, t_m, t_s + p, \ldots, t_s + p)$ is a relevant combination of processors variables and we can apply the recursion.

- $F(t_1, \ldots, t_m, i_1, \ldots, i_m)$ equals the minimum of F_s^* over all possible sizes s of the tasks of the sub-problem (line 21).

There are $O(n^{2m})$ possible tuples of processors variables and $O(n^m)$ tuples of tasks variables to consider. So we have $O(n^{3m})$ combinations of variables for which F has to be computed. Since m is fixed, for each combination (lines 5–24), F is computed in constant time. Hence, the algorithm runs in $O(n^{3m})$.

4.1.4 Open Questions

We have shown that $Pm|r_i, p_i = p, \mathit{size}_i| \sum C_i$ can be solved in polynomial time. The same algorithm can be used to minimize $C_{\max}$. We do not think that our algorithm is likely to be generalized to handle other criteria and so, the complexity status of the weighted completion time problem is still an open question.

4.2 Scheduling Tall/Small Tasks, $P|r_i, p_i = 1, \mathit{size}_i \in \{1, m\}|L_{\max}$

We study the scheduling situation where n tasks, subjected to release dates and due dates, have to be scheduled on m parallel processors. We show that, when tasks have unit processing times and either require 1 or m processors simultaneously, the minimum maximal lateness can be computed in polynomial time. Two algorithms are described. The first one is based on a linear programming formulation of the problem while the second one is a combinatorial algorithm. The complexity status of this "tall/small" task scheduling problem $P|r_i, p_i = 1, \mathit{size}_i \in \{1, m\}|L_{\max}$ was unknown before, even for 2 processors.

When all release dates are equal, the arbitrary size problem for any *fixed* number of processors, denoted $Pm|p_i = 1, \mathit{size}_i|T_{\max}$, can be solved in polynomial time [50]. Similarly, the problem for any *fixed* number of sizes, denoted $P|p_i = 1, |\{\mathit{size}_i\}| = c|T_{\max}$, can be solved in polynomial time. Both problem can be solved using dynamic programming. However, the arbitrary size problem is NP-Hard in the strong sense [160].

Algorithm 4 Computation of the values $F(t_1,\dots,t_m,i_1,\dots,i_m)$ ($F = C_{\max}$)

```
$IVARS \rightarrow$ list of tasks variables sorted in dec. lexicographical order
$TVARS \rightarrow$ list of processors variables sorted in dec. lexicographical order
for $(i_1,\dots,i_m)$ in $IVARS$ do
  for $(t_1,\dots,t_m)$ in $TVARS$ do
    if $\forall s \in \{1,\dots,m\}, N(i_s,s) < 0$ then
      $F(t_1,\dots,i_1,\dots,i_m) = t_m$
    else
      $F(t_1,\dots,i_1,\dots,i_m) = \infty$
      for $s \in \{1,\dots,m\}$ do
        if $N(i_s,s) \geq 0$ then
          $x \rightarrow N(i_s,s)$
          if $t_1 < r_x$ then
            $F^*_s \rightarrow F(\max(t_1,r_x),\dots,\max(t_m,r_x),i_1,\dots,i_m)$
          else
            if $t_s + p \in \Theta$ then
              $F^*_s \rightarrow F(t_{s+1},\dots,t_m,t_s+p,\dots,t_s+p,i_1,\dots,i_{s-1},x+1,i_{s+1},\dots,i_m)$
            else
              $F^*_s \rightarrow \infty$
            end if
          end if
          $F(t_1,\dots,i_1,\dots,i_m) \rightarrow \min(F^*_s, F(t_1,\dots,i_1,\dots,i_m))$
        end if
      end for
    end if
  end for
end for
```

When preemption is allowed, even with arbitrary processing times and release dates, the very general problem $Pm|r_i, size_i, pmtn|T_{\max}$ problem problem is easy to solve. Existing algorithms are based on a Linear Programming formulation where a variable is associated to each subset of tasks whose total resource requirement is less than m (see for instance [35]). Unfortunately, there are some instances of $P|r_i, p_i = 1, size_i \in \{1, m\}|L_{\max}$ for which the non-preemptive maximum lateness is strictly larger than the preemptive maximum lateness. In the preemptive schedule of Figure 4.4, $L_{\max} = 0$, while the value of $L_{\max}$ is at least 1 for any non-preemptive schedule.

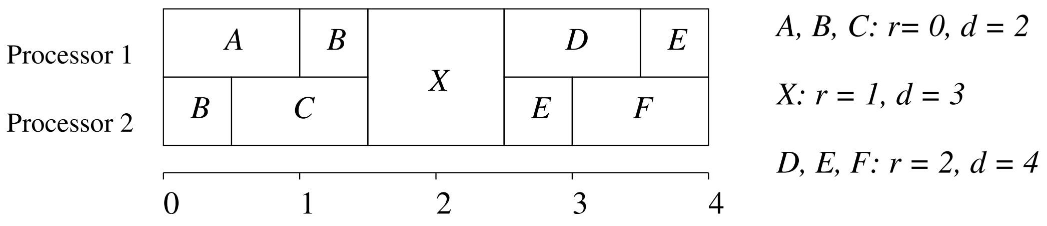


Figure 4.4: An optimal preemptive schedule.

In the following, we focus on the decision variant of the maximal lateness problem. For a fixed value of $L_{\max}$, it is easy to compute a deadline $\bar{d_i} = d_i + L_{\max}$ for each task i and a schedule is said to be feasible if tasks are completed before their deadlines. To compute the minimal maximal lateness, one can find the smallest value of $L_{\max}$ for which there is a feasible schedule. Since one can easily build a feasible schedule with $L_{\max} = n$, there are no more than n values to test. A dichotomic search could be used to reduce the number of iterations. So, the $L_{\max}$ problem can be solved in polynomial time provided that the deadline scheduling problem is solvable in polynomial time.

In Section 4.2.1, we introduce a linear programming formulation of the problem and in Section 4.3 we describe a combinatorial algorithm to solve the problem. Finally we draw some conclusions in Section 4.3.1.

4.2.1 An LP Formulation

In Section 4.2.2 we describe some linear constraints that must be met by any feasible schedule. In Section 4.2.3 we show that if these constraints hold, we can build a preemptive schedule of tall tasks that "implicitly" takes into account the small tasks. This schedule is transformed in Section 4.2.4 into a non-preemptive schedule of both small and tall tasks.

4.2.2 Necessary Conditions

To simplify the presentation of the algorithm we will introduce some time indexed variables. There are few relevant time points so the total number of variables remains polynomial in n. Indeed, we can assume that the distance between two consecutive release dates r_x, r_y is not larger than n (otherwise, the tasks could be split in two subsets $\{i : r_i \leq r_x\}$ and $\{i : r_i \geq r_y\}$ that could be scheduled independently). On top of that, we can also assume that the largest deadline is not larger than the largest release date plus n. Due to these assumptions, we have a polynomial number of relevant integer time points t to consider. In the following, unless precisely stated, time points and time slots are integral.

Consider a feasible schedule and, for each tall task i and for each integer time point, let t, $x_i^t \in [0, 1]$ be the total time during which i executes in the time-slot $[t, t+1)$. Each tall task has to be scheduled somewhere between its release date and its deadline so,

$$\forall i \in \mathcal{T}_m, \sum_{t=r_i}^{\bar{d}_i - 1} x_i^t = 1. \tag{4.4}$$

On top of that, the total time during which tall tasks are processed in a single time slot does not exceed the size of the time slot, *i.e.*,

$$\forall t, \sum_{i \in \mathcal{T}_m} x_i^t \leq 1. \tag{4.5}$$

Now let us focus on small tasks. For any time interval $[t_1, t_2)$, let $\mathcal{T}_1(t_1, t_2)$ be the set of small tasks that have to execute in the time interval $[t_1, t_2)$, *i.e.*,

$$\mathcal{T}_1(t_1, t_2) = \{i \in \mathcal{T}_1 : t_1 \leq r_i < \bar{d}_i \leq t_2\}.$$

Note that in a non-preemptive schedule, q small tasks cannot be scheduled in less than $\lceil \frac{q}{m} \rceil$ time units. Since in a time interval $[t_1, t_2)$ there are only $t_2 - t_1 - \sum_{i \in \mathcal{T}_m} \sum_{t=t_1}^{t_2-1} x_i^t$ time units available to schedule tall tasks, we get

$$\forall [t_1, t_2), \sum_{i \in \mathcal{T}_m} \sum_{t=t_1}^{t_2-1} x_i^t + \left\lceil \frac{|\mathcal{T}_1(t_1, t_2)|}{m} \right\rceil \leq t_2 - t_1. \tag{4.6}$$

Hence, if there is feasible schedule, there is a feasible solution of the Linear Program (4.7).

$$\begin{cases} \forall i \in \mathcal{T}_m, \sum_{t=r_i}^{\bar{d}_i - 1} x_i^t = 1 \\ \forall t, \sum_{i \in \mathcal{T}_m} x_i^t \leq 1 \\ \forall [t_1, t_2), \sum_{i \in \mathcal{T}_m} \sum_{t=t_1}^{t_2-1} x_i^t + \left\lceil \frac{|\mathcal{T}_1(t_1, t_2)|}{m} \right\rceil \leq t_2 - t_1 \\ \forall i \in \mathcal{T}_m, \forall t, x_i^t \geq 0 \end{cases} \tag{4.7}$$

In the following, we show that a feasible schedule exists if there is a feasible solution of (4.7).

4.2.3 Preemptive Schedule of Tall Tasks

From now on, we assume that tasks $1, \ldots, |\mathcal{T}_m|$ are the tall ones and that they are sorted in non-decreasing order of due dates, *i.e.*, $d_1 \leq \ldots \leq d_{|\mathcal{T}_m|}$.

A solution x of (4.7), specifies the duration x_i^t a tall task i is scheduled in $[t, t+1)$. To precisely build a preemptive schedule of tall tasks, it remains to decide how pieces of tall tasks are scheduled inside each time slot $[t, t+1)$. Let $\mathcal{S}(x)$ be the schedule where, in each time slot, pieces of tall tasks are scheduled from left to right according to their initial numbering (*i.e.*, in non-decreasing order of deadlines). Now let us consider the solution $\bar{x}$ that lexicographically minimizes the vector of average completion times $(C_1, \ldots, C_{|\mathcal{T}_m|})$, where $C_i = \sum_{t=r_i}^{\bar{d}_i - 1} t \cdot x_i^t$. The next proposition shows that in any such solution x_i^t is either 0 or 1.

Proposition 25. *In $\mathcal{S}(\bar{x})$ tall tasks are not preempted and they start at integer time points.*

Proof. Let k be the first task for which the proposition does not hold (all tasks with smaller indices are not preempted and start at integer time points). Let $[t, t+1)$ and $[t', t'+1)$ be the time slots in which k respectively starts and is completed in $\mathcal{S}(\bar{x})$.
First, we show that in $[t, t+1)$, k is the only tall task to execute. Indeed, if there was another tall task l that executes there, we would have $l > k$ (because of our assumption on k). Therefore, we could exchange a small piece of l that executes in $[t, t+1)$ with a small piece of k that executes in $[t', t'+1)$. In other terms, we could build a vector $\hat{x}$ that equals $\bar{x}$ except for the following values:

$$\begin{cases} \hat{x}_k^t = \bar{x}_k^t + \varepsilon \\ \hat{x}_k^{t'} = \bar{x}_k^{t'} - \varepsilon \\ \hat{x}_l^{t'} = \bar{x}_l^{t'} + \varepsilon \\ \hat{x}_l^t = \bar{x}_l^t - \varepsilon \end{cases},$$

where $\varepsilon = \min\{\bar{x}_k^{t'}, \bar{x}_l^t\} > 0$. In the resulting schedule $\mathcal{S}(\hat{x})$ the average completion time of k is smaller than its average completion time in $\mathcal{S}(\bar{x})$ and task l is completed before its deadline (because $\bar{d}_l \geq \bar{d}_k$). Moreover, the "load" of each time slot $[\tau, \tau+1)$ is the same in both schedule, *i.e.*,

$$\forall \tau, \sum_{i \in \mathcal{T}_m} \bar{x}_i^\tau = \sum_{i \in \mathcal{T}_m} \hat{x}_i^\tau.$$

It follows that $\hat{x}$ is a feasible solution of (4.7) and it is smaller lexicographically than $\bar{x}$. This contradicts our hypothesis on $\bar{x}$.
Second, note that some constraints of (4.7) must be tight for $\bar{x}$ and prevent us from increasing $\bar{x}_k^t$ by ε and decrease $\bar{x}_k^{t'}$ by ε. Indeed, if we could perform this exchange, we would obtain a "lower" feasible solution in the lexicographical order we defined. Constraints (4.4) do not prevent us from making this exchange. Neither do Constraints (4.5) since k is the only tall task to execute in time slot t. Now, let us examine Constraints (4.6). We are going to show that a slightly more complex exchange is possible. In the following, the notation $\mathcal{I}(t_1, t_2)$ refers to the constraint (4.6) over the interval $[t_1, t_2)$. Let Ω be the set of constraints (4.6) with $t_1 \leq t$ and $t < t_2 \leq t'$ that are tight for $\bar{x}$. It is easy to see that Constraints (4.6) that do not belong to Ω do not prevent us from increasing $\bar{x}_k^t$. Among the constraints of Ω let us pick one with maximum t_1. Since the constraint is tight, we have

$$\sum_{i \in \mathcal{T}_m} \sum_{t=t_1}^{t_2-1} \bar{x}_i^t = t_2 - t_1 - \left\lceil \frac{|\mathcal{T}_1(t_1, t_2)|}{m} \right\rceil.$$

Hence, $\sum_{i \in \mathcal{T}_m} \sum_{t=t_1}^{t_2-1} \bar{x}_i^t$ takes an integer value and consequently, there is another tall task u that is partially executed between t_1 and t_2. If u were partially executed between t and t_2 then we could exchange a piece of it with the last piece of k that executes in $[t', t'+1)$ (we have $\bar{d}_u \geq \bar{d}_k$ because u is preempted and so the exchange is feasible). We would have decreased the average completion time of k; which would contradict our initial hypothesis. So u is partially executed in a time slot $[\tau, \tau+1)$ between t_1 and t. Let $\tilde{x}$ be the vector that equals $\bar{x}$ except of the following values:

$$\begin{cases} \tilde{x}_u^\tau = \bar{x}_u^\tau - \varepsilon \\ \tilde{x}_u^{t'} = \bar{x}_u^{t'} + \varepsilon \\ \tilde{x}_k^{t'} = \bar{x}_k^{t'} - \varepsilon \\ \tilde{x}_k^t = \bar{x}_k^t + \varepsilon \end{cases},$$

for a small positive value ε. Note that a piece of u can be scheduled at t' because $\bar{d}_u \geq \bar{d}_k$. Moreover, the only constraints that could be violated by the exchange are those in the set Ω (the value of $\tilde{x}_k^t$ is consistent with (4.5) because k is the only tall task that executes in $[t, t+1)$). Let $\mathcal{I}(t_1', t_2')$ be a violated constraint of Ω. Because of our hypothesis on t_1, we have $t_1' \leq t_1$. Hence, it is easy to verify that the load induced by the tall tasks between t_1' and t_2' does not increase. Therefore, the resulting schedule $\mathcal{S}(\tilde{x})$ is lower than $\mathcal{S}(\bar{x})$ in the lexicographical order. This contradicts our hypothesis. □

4.2.4 From Preemptive to Non-Preemptive Schedules

In Section 4.2.3, we have shown that there is a solution $\bar{x}$ of (4.7) such that in $\mathcal{S}(\bar{x})$, tall tasks are not preempted and start at integer time points. In Proposition 26 we show that small tasks can be scheduled in $\mathcal{S}(\bar{x})$ too.

Proposition 26. *Small tasks can be scheduled in* $\mathcal{S}(\bar{x})$.

Proof. Let us sort small tasks in non-decreasing order of deadlines and let us add them one after the other into $\mathcal{S}(\bar{x})$. Each time, the current task starts at the first time point after its release date where one processor is available. Note that because tall tasks are not preempted and start at integer time points, small tasks are not preempted either and also start at integer time points.
Let k be the first small task that is completed after its deadline and let t be the earliest time point such that all processors are full between t and $\bar{d}_k$. Note that since k is not completed by its deadline $t \leq r_k$. Let Ψ be the set of small tasks that are scheduled in $[t, \bar{d}_k)$. Tasks in Ψ have a release

date greater than or equal to t (otherwise they would be scheduled in $[t-1,t)$) and a deadline smaller than or equal to $\bar{d}_k$ (because tasks are sorted in non-decreasing order of deadlines). Since all processors are full, there are exactly $m(\bar{d}_k - t - \sum_{i \in \mathcal{T}_m} \sum_{t'=t}^{\bar{d}_k - 1} \bar{x}_i^t)$ tasks in Ψ. On top of that, $\Psi \subset \mathcal{T}_1(t, \bar{d}_k)$ and $k \notin \Psi$ but $k \in \mathcal{T}_1(t, \bar{d}_k)$. Hence,

$$\mathcal{T}_1(t, \bar{d}_k) > m(\bar{d}_k - t - \sum_{i \in \mathcal{T}_m} \sum_{t'=t}^{\bar{d}_k - 1} \bar{x}_i^t).$$

This contradicts (4.6). □

4.3 A Combinatorial Algorithm

For each time interval $[t_1, t_2)$, we define recursively the *slack* of the interval, denoted $\mathit{\Delta}(t_1, t_2)$. The slacks can be computed in polynomial time and we show later that there exists a feasible schedule if and only if all the intervals have non-negative slack. Intuitively, $\mathit{\Delta}(t_1, t_2)$ is an upper bound on the number of *free* time slots in $[t_1, t_2)$ in any schedule of the tasks in $\mathcal{T}(t_1, t_2)$. (For a given schedule a free time slot is a time unit in which all the m processors are idle.) To define the slack we define *static* slack and *dynamic* slack.

Definition 14. *For any interval $[t_1, t_2)$, with $t_2 > t_1 + 1$, the slack $\mathit{\Delta}(t_1, t_2)$ is the minimum of the static slack $\mathit{\Delta}_s(t_1, t_2)$ and of the dynamic slack $\mathit{\Delta}_d(t_1, t_2)$. For any interval $[t_1, t_2)$, with $t_1 \le t_2 \le t_1 + 1$, $\mathit{\Delta}(t_1, t_2) = \mathit{\Delta}_s(t_1, t_2)$.*

Definition 15. *The static slack over $[t_1, t_2)$, $\mathit{\Delta}_s(t_1, t_2)$, equals*

$$t_2 - t_1 - \left(|\mathcal{T}_m(t_1, t_2)| + \left\lceil \frac{|\mathcal{T}_1(t_1, t_2)|}{m} \right\rceil \right)$$

Definition 16. *The dynamic slack over $[t_1, t_2)$, with $t_2 > t_1 + 1$, $\mathit{\Delta}_d(t_1, t_2)$, is the minimum of*

$$\mathit{\Delta}(t_1, t_2') + \mathit{\Delta}(t_1', t_2) - |\mathcal{T}_m(t_1, t_2) \setminus (\mathcal{T}_m(t_1, t_2') \cup \mathcal{T}_m(t_1', t_2))|$$

over t_1', t_2' with $t_1 < t_1' \le t_2' < t_2$.

The next proposition shows that if there is a feasible schedule, then all intervals have non-negative slack.

Proposition 27. *There are at most $\mathit{\Delta}(t_1, t_2)$ idle time slots between t_1 and t_2 in any feasible schedule of $\mathcal{T}(t_1, t_2)$.*

Proof. Consider an interval $[t_1, t_2)$ such that there is a feasible schedule of $\mathcal{T}(t_1, t_2)$ with more than $\mathit{\Delta}(t_1, t_2)$ idle time slots and assume that $t_2 - t_1$ is minimal among such intervals. There are $|\mathcal{T}_m(t_1, t_2)|$ tall tasks and $|\mathcal{T}_1(t_1, t_2)|$ small tasks and at least $\left(|\mathcal{T}_m(t_1, t_2)| + \left\lceil \frac{|\mathcal{T}_1(t_1, t_2)|}{m} \right\rceil \right)$ time units are required to schedule these tasks. It follows that the number of idle time slots in a feasible schedule is no more than the static slack $\mathit{\Delta}_s(t_1, t_2)$. Hence the slack equals the dynamic slack; that is, $\mathit{\Delta}(t_1, t_2) = \mathit{\Delta}_d(t_1, t_2)$. Suppose that $\mathit{\Delta}_d(t_1, t_2)$ is determined by the two time points t_1' and t_2'; that is, $\mathit{\Delta}(t_1, t_2) = \mathit{\Delta}(t_1, t_2') + \mathit{\Delta}(t_1', t_2) - |\mathcal{T}_m(t_1, t_2) \setminus (\mathcal{T}_m(t_1, t_2') \cup \mathcal{T}_m(t_1', t_2))|$. Consider the feasible schedule $\mathcal{S}$ of $\mathcal{T}(t_1, t_2)$ and let $\mathcal{S}_r$ and $\mathcal{S}_l$ be the restriction of $\mathcal{S}$ to the tasks of $\mathcal{T}(t_1, t_2')$ and to the tasks of $\mathcal{T}(t_1', t_2)$. The number of idle time slots in $\mathcal{S}$ is less than or equal to the number of those in $\mathcal{S}_r$ plus the number of those in $\mathcal{S}_l$. Given our hypothesis on $t_2 - t_1$ this is lower than or equal to $\mathit{\Delta}(t_1, t_2') + \mathit{\Delta}(t_1', t_2)$. On top of that, large tasks in $\mathcal{T}_m(t_1, t_2) \setminus (\mathcal{T}_m(t_1, t_2') \cup \mathcal{T}_m(t_1', t_2))$ are not scheduled in $\mathcal{S}_l$ nor in $\mathcal{S}_r$ so the number of idle time slots in $\mathcal{S}$ is less than or equal to

$$\mathit{\Delta}(t_1, t_2') + \mathit{\Delta}(t_1', t_2) - |\mathcal{T}_m(t_1, t_2) \setminus (\mathcal{T}_m(t_1, t_2') \cup \mathcal{T}_m(t_1', t_2))|,$$

which equals $\mathit{\Delta}(t_1, t_2)$. This contradicts our initial hypothesis. □

The more complicated part is to show that if there is no feasible schedule then there must be at least one interval with negative slack. In order to prove this we first prove two propositions. In the proofs we consider several instances simultaneously. Given an instance I, $\Delta(I,t_1,t_2)$, $\mathcal{T}(I,t_1,t_2)$, $\mathcal{T}_1(I,t_1,t_2)$ and $\mathcal{T}_m(I,t_1,t_2)$ denote the slack and the sets associated to I for a given interval $[t_1,t_2)$. From now on, to simplify notation we also assume that the smallest release time is 0.

Proposition 28. *Consider an instance I, and let J be and instance given by removing a tall task i, with $r_i = 0$, if such exists. For all $t < \bar{d}_i$, $\Delta(J,0,t) = \Delta(I,0,t)$, and for all $t \geq \bar{d}_i$, $\Delta(J,0,t) = \Delta(I,0,t)+1$.*

Proof. The first part follows directly from the definition of slack. Consider the second part and to obtain a contradiction assume that the equality does not hold. Let t be the minimum time such that that $\Delta(J,0,t) \neq \Delta(I,0,t)+1$. We first show that $\Delta(J,0,t) \leq \Delta(I,0,t)+1$. We consider two cases.
Case 1: $\Delta(I,0,t) = \Delta_s(I,0,t)$. Since $\Delta_s(J,0,t) = \Delta_s(I,0,t)+1$, we have $\Delta(J,0,t) \leq \Delta(I,0,t)+1$.
Case 2: $\Delta(I,0,t) = \Delta_d(I,0,t)$. Let t_1, t_2 be two time points that determine $\Delta_d(I,0,t)$. Note that $\Delta(I,t_1,t) = \Delta(J,t_1,t)$ (because $0 < t_1$). If $\bar{d}_i \leq t_2$ then i does not belong to $\mathcal{T}_m(I,0,t) \setminus (\mathcal{T}_m(I,0,t_2) \cup \mathcal{T}_m(I,t_1,t))$. Hence, the slack of J is not larger than

$$\Delta(J,0,t_2) + \Delta(I,t_1,t) - |\mathcal{T}_m(I,0,t) \setminus (\mathcal{T}_m(I,0,t_2) \cup \mathcal{T}_m(I,t_1,t))|\,.$$

Since $t_2 < t$, by our assumption $\Delta(J,0,t_2) = \Delta(I,0,t_2)+1$. Hence, $\Delta(J,0,t) \leq \Delta(I,0,t)+1$. If $\bar{d}_i > t_2$ then

$$\begin{aligned}\Delta(J,0,t) &\leq \Delta(I,0,t_2) + \Delta(I,t_1,t) - (|\mathcal{T}_m(I,0,t) \setminus (\mathcal{T}_m(I,0,t_2) \cup \mathcal{T}_m(I,t_1,t))| - 1)\\ &= \Delta(I,0,t)+1.\end{aligned}$$

Similarly, it can be shown that $\Delta(I,0,t) \leq \Delta(J,0,t)-1$, by considering the two cases $\Delta(J,0,t) = \Delta_s(J,0,t)$ and $\Delta(J,0,t) = \Delta_d(J,0,t)$. □

Proposition 29. *Consider an instance I. Let k be the number of small tasks in I with release date 0. Let J be the instance derived from I by removing $\min\{k,m\}$ small tasks with the smallest deadline among tasks with release date 0, and by changing the release date of the rest of the tasks with release date 0 (if any) to 1. Let x be a tall task in I with the earliest deadline among tall tasks with release date 0 (if such exists), and let t be $\bar{d}_x$ if x is exists and ∞ otherwise. For all $1 < t_2 < t$,*

$$\min\{\Delta(I,0,t_2), \Delta(I,1,t_2)\} \leq \Delta(J,1,t_2).$$

Proof. To obtain a contradiction assume that t_2 is the minimum time for which the inequality does not hold. If $k \leq m$ then since there are no tall tasks with release date 0 and deadline at most t_2 in I, we have $\Delta(I,1,t_2) = \Delta(J,1,t_2)$; a contradiction. Suppose that $k > m$. Let $\bar{d}$ be the earliest deadline of a task in I with release date 0 that was not removed. If $t_2 < \bar{d}$ then again $\mathcal{T}(J,1,t_2)$ does not contain any task with release time 0 in I, and thus, $\Delta(I,1,t_2) = \Delta(J,1,t_2)$; a contradiction. The only remaining case is $t_2 \geq \bar{d}$. We consider two cases.
Case 1: $\Delta(J,1,t_2) = \Delta_s(J,1,t_2)$. Since $t_2 \geq \bar{d}$ $|\mathcal{T}_1(J,1,t_2)| = |\mathcal{T}_1(I,0,t_2)| - m$ and thus

$$\left\lceil \frac{|\mathcal{T}_1(J,1,t_2)|}{m} \right\rceil = \left\lceil \frac{|\mathcal{T}_1(I,0,t_2)|}{m} \right\rceil - 1.$$

Recall also that since no tall task in I has release date 0 and deadline at most t_2 we have for all $1 < s \leq t_2$, $\mathcal{T}_m(I,0,s) = \mathcal{T}_m(J,1,s)$. We get

$$\begin{aligned}\Delta(J,1,t_2) &= t_2 - 1 - (|\mathcal{T}_m(J,1,t_2)| + \left\lceil \frac{|\mathcal{T}_1(J,1,t_2)|}{m} \right\rceil)\\ &= t_2 - (|\mathcal{T}_m(I,0,t_2)| + \left\lceil \frac{|\mathcal{T}_1(I,0,t_2)|}{m} \right\rceil)\\ &= \Delta_s(I,0,t_2) \geq \Delta(I,0,t_2).\end{aligned}$$

Case 2: $\Delta(J,1,t_2) = \Delta_d(J,1,t_2)$. Let t_1', t_2' with $1 < t_1' \le t_2' < t_2$ be time points that determine the dynamic slack. Recall that by our assumption $\min\{\Delta(I,0,t_2'), \Delta(I,1,t_2')\} \le \Delta(J,1,t_2')$. Since $t_1' > 1$, $\Delta(I,t_1',t_2) = \Delta(J,t_1',t_2)$. We get

$$\begin{aligned}
\Delta(J,1,t_2) &= \Delta(J,1,t_2') + \Delta(J,t_1',t_2) \\
&\quad - |\mathcal{T}_m(J,1,t_2) \setminus (\mathcal{T}_m(J,1,t_2') \cup \mathcal{T}_m(J,t_1',t_2))| \\
&\ge \min\{\Delta(I,0,t_2'), \Delta(I,1,t_2')\} + \Delta(I,t_1',t_2) \\
&\quad - |\mathcal{T}_m(I,0,t_2) \setminus (\mathcal{T}_m(I,0,t_2') \cup \mathcal{T}_m(I,t_1',t_2))| \\
&\ge \min\{\Delta(I,0,t_2), \Delta(I,1,t_2)\}\,.
\end{aligned}$$

□

Proposition 30. *If all time intervals have non-negative slack then there is a feasible schedule.*

Proof. We prove that if there is no feasible schedule then there must be an interval with a negative slack. Define the *size* of an instance I as $\sum_{\mathcal{T}} \bar{d}_i - r_i$. We prove it by induction on the size of the instance. The proposition clearly holds for instances of size 1 since such instances consist of one task. Suppose that the proposition holds for all instances of size at most s. We prove it for instances of size $s+1$. Consider an instance I of size $s+1$ with no feasible schedule. If I has both a tall task and a small task with release date 0 and deadline 1 or more than m small tasks with release time 0 and deadline 1 then $\Delta(I,0,1) < 0$. Suppose that this is not the case. Let k be the number of small tasks in I with release date 0. Consider two instances J_S and J_T defined as follows. The instance J_S is derived from I by removing $\min\{k,m\}$ small tasks with the smallest deadline among tasks with release date 0, and by changing the release date of the rest of the tasks with release date 0 (if any) to 1. The instance J_T is derived from I by removing a tall task x with the smallest deadline among tall tasks with release date 0, and by changing the release date of the rest of the tasks with release date 0 (if any) to 1. We assume that the task x exists. Note that if any of J_T and J_S has a feasible schedule then this schedule can be extended to a feasible schedule for I. We conclude that both J_S and J_T have no feasible schedule. Since the size of both J_S and J_T is at most s, there is an interval with negative slack in both J_S and J_T. If for $J \in \{J_S, J_T\}$, there exists $1 < t_1 < t_2$ such that $\Delta(J,t_1,t_2) < 0$ then since $\Delta(I,t_1,t_2) = \Delta(J,t_1,t_2)$ we are done. Suppose that this is not the case. Hence, there are $t_S, t_T > 1$ such that $\Delta(J_S,1,t_S) < 0$ and $\Delta(J_T,1,t_T) < 0$. We consider several cases.
Case 1: $t_T \ge \bar{d}_x$. Let I' be the instance given by removing task x from I. Since the sets of tasks in J_T and I' are the same $\Delta(I',0,t_T) \le \Delta(J_T,1,t_T)+1$. By Proposition 28 $\Delta(I,0,t_T) = \Delta(I',0,t_T)-1$. Hence, $\Delta(I,0,t_T) \le \Delta(J_T,1,t_T) < 0$.
Case 2: $t_S < \bar{d}_x$. By Proposition 29 $\min\{\Delta(I,0,t_S), \Delta(I,1,t_S)\} \le \Delta(J_S,1,t_S) < 0$.
Case 3: The remaining case is $t_S \ge \bar{d}_x$ and $t_T < \bar{d}_x$. Let l be the number of large tasks in I with release date 0 and deadline at most t_S. Let J_S' be the instance given by removing these l tasks from J_S. By proposition 28 $\Delta(J_S',1,t_S) = \Delta(J_S,1,t_S) + l < l$. Let I' be the instance given by removing these l tasks from I. Note that $\Delta(I,1,t_S) = \Delta(I',1,t_S)$. By Proposition 29 $\min\{\Delta(I',0,t_S), \Delta(I',1,t_S)\} \le \Delta(J_S',1,t_S) < l$. If $\Delta(I',0,t_S) < l$ then by Proposition 28 $\Delta(I,0,t_S) = \Delta(I',0,t_S) - l < 0$. Otherwise, $\Delta(I,1,t_S) < l$. We get that

$$\begin{aligned}
\Delta(I,0,t_S) &\le \Delta(I,0,t_T) + \Delta(I,1,t_S) - |\mathcal{T}_m(I,0,t_S) \setminus (\mathcal{T}_m(I,0,t_T) \cup \mathcal{T}_m(I,1,t_S))| \\
&< 0 + l - l = 0
\end{aligned}$$

To conclude the proof, we just have to consider the situation where the task x does not exist (*i.e.*, there is no tall task with release date 0). In such a case, the same reasoning as for Case 2 applies. □

4.3.1 Open Questions

We have presented two algorithms for scheduling tall/small multiprocessor tasks with unit processing time to minimize maximum tardiness. The first one relies on linear programming and the second one is purely combinatorial. An interesting open question is whether one of them could be extended to solve a larger class of problem such as $Pm|r_i, p_i = 1, size_i|T_{\max}$.

Chapter 5

Batching Equal Length Jobs

We study the situation where n jobs $\{J_1, \ldots, J_n\}$ have to be scheduled on a batching machine. Jobs cannot start before their release dates and all jobs of the same batch start and are completed simultaneously, *i.e.*, at the starting time (respectively at the completion time) of the batch. Two types of batching machines are studied.

- On a serial batching machine, the length of a batch equals the sum of the processing times of its jobs. When a new batch starts, a constant setup time s occurs.

- On a parallel batching machine, there are at most b jobs per batch and the length of a batch is the largest processing time of its jobs. Two situations are often distinguished. The bounded case with $b < n$ and the unbounded case with $b = n$. In this environment, no setup is assumed. However, a constant setup time s could be easily taken into account by increasing of s the processing time of each job.

Following the notation of [47], these problems are denoted respectively by $1|s\text{-}batch, r_i|F$, $1|p\text{-}batch, b < n, r_i|F$ (bounded case) and $1|p\text{-}batch, r_i|F$ (unbounded case). We refer to [6], [39], [47], [93], [168], [187], [217] for extended reviews on pure batch scheduling problems and on extensions (*e.g.* scheduling group of jobs with group-dependent setup times, jobs requiring several machines throughout their execution, *etc.*). Complexity results for problems with identical release dates are summarized in Table 5. The problems $1|r_i|F$ are NP-Hard for $L_{\max}, \sum C_i, \sum T_i$, hence the corresponding batching problems $1|s\text{-}batch, r_i|F$ and $1|p\text{-}batch, r_i|F$ are also NP-Hard.

These results leave open the status of most of the problems with arbitrary release dates and equal processing times. In this chapter, we first study (Section 5.1) the parallel unbounded problem and we show that, for arbitrary processing times, an optimum schedule can be computed in pseudopolynomial time for any regular objective function. When processing times are equal, the algorithm runs in polynomial time. We then show (Section 5.2) that serial and (bounded) parallel batching problems can be solved in polynomial time for the class of **ordered objective functions** (see Definition 2).

5.1 Parallel Unbounded Batching, $1|p\text{-}batch, r_i|F$

We study the parallel unbounded problem $1|p\text{-}batch, r_i|F$. From now on, we assume that $F = \sum f_i$ is regular and that jobs are ordered in non-decreasing order of processing times.

Proposition 31. *There is an optimal schedule such that, for any batch, the jobs that are smaller than the length of the batch and that are released before or at the starting time of the batch are either executed in the batch or in a previous batch.*

Proof. Consider a job i and a batch B for which the proposition does not hold. The job i can be moved to B and since F is regular, this does not increase the value of objective function. $\square$

Problem	Complexity	References
$1\|p\text{-}batch, b < n\|C_{\max}$	$O(n \log n)$	[39]
$1\|s\text{-}batch\|L_{\max}$	$O(n^2)$	[217]
$1\|p\text{-}batch\|L_{\max}$	$O(n^2)$	[39]
$1\|p\text{-}batch, b < n\|L_{\max}$	Unary NP-Hard	[39]
$1\|s\text{-}batch\|\sum U_i$	$O(n^3)$	[49]
$1\|s\text{-}batch\|\sum w_iU_i$	binary NP-Hard	[49]
$1\|s\text{-}batch, p_i = p\|\sum w_iU_i$	$O(n^4)$	[120]
$1\|p\text{-}batch\|\sum U_i$	$O(n^3)$	[39]
$1\|p\text{-}batch\|\sum w_iU_i$	binary NP-Hard	[39]
$1\|p\text{-}batch, b < n\|\sum U_i$	Unary NP-Hard	[39]
$1\|s\text{-}batch\|\sum C_i$	$O(n \log n)$	[76]
$1\|s\text{-}batch, p_i = p\|\sum C_i$	polynomial in $\log p, \log s, \log n$	[199]
$1\|s\text{-}batch\|\sum w_iC_i$	unary NP-Hard	[6]
$1\|s\text{-}batch, p_i = p\|\sum w_iC_i$	$O(n \log n)$	[6]
$1\|p\text{-}batch\|\sum w_iC_i$	$O(n \log n)$	[39]
$1\|p\text{-}batch, b < n\|\sum C_i$	$O(n^{b(b-1)})$	[39]
$1\|s\text{-}batch\|\sum T_i$	binary NP-Hard	[91]
$1\|s\text{-}batch\|\sum T_i$	$O(n^{11}\max(\max_i p_i, s)^7)$	[18]
$1\|p\text{-}batch\|\sum w_iT_i$	binary NP-Hard	[39]
$1\|p\text{-}batch, b < n\|\sum T_i$	unary NP-Hard	[39]

Table 5.1: Overview of the complexity results

Proposition 31 suggests that once the starting time of the largest batch (the one that contains the largest job n) is known, we can decompose the problem into two sub-problems that occur before and after this batch.

More formally, a state of the dynamic program is defined by (1) the index k of the largest job considered in the problem, (2) a right fictive batch starting at s with length π_s and (3) a left fictive batch starting at e with length $\pi_e \geq p_k$. In the recursion we try to schedule the jobs i with $i \leq k$ and $s < r_i \leq e$ between $s + \pi_s$ and $e + \pi_e$ (jobs can be scheduled either in the right fictive batch or between the fictive batches). Let $F(k, s, \pi_s, e, \pi_e)$ be the minimal cost of such a schedule.

Proposition 32. *If $r_k > e$ or if $r_k \leq s$ then $F(k, s, \pi_s, e, \pi_e) = F(k-1, s, \pi_s, e, \pi_e)$ otherwise, $F(k, s, \pi_s, e, \pi_e)$ is the minimum of $F(k-1, s, \pi_s, e, \pi_e) + f_k(e + \pi_e)$ and of*

$$\min_{t \in [\max(s+\pi_s, r_k), e-p_k]} F(k, s, \pi_s, t, p_k) + F(k, t, p_k, e, \pi_e).$$

Proof. If $r_k > e$ or if $r_k \leq s$ then k is not considered in the sub-problem and hence, $F(k, s, \pi_s, e, \pi_e) = F(k-1, s, \pi_s, e, \pi_e)$. Now assume that $s < r_i \leq e$ and consider the non-dominated schedule that realizes $F(k, s, \pi_s, e, \pi_e)$. The job k is either scheduled in the right fictive batch or it is scheduled in a batch that starts after or at s and that is completed before or at e.

- If k is scheduled in the fictive batch, then the cost of scheduling k is $f_k(e + \pi_e)$ and all other jobs have to be scheduled between $s + \pi_s$ and e or in the fictive batch. So, $F(k, s, \pi_s, e, \pi_e) \geq F(k-1, s, \pi_s, e, \pi_e) + f_k(e + \pi_e)$.

- If k is scheduled in a batch B starting at t between s and e then, we know that the batch length is p_k (since k is the largest job). Moreover, it has to be completed before e and it cannot start in the left fictive batch so, $t \in [\max(s + \pi_s, r_k), e - p_k]$. According to our dominance property, all other jobs are either executed in batch B or in a previous batch or are released after t. Jobs released after t cannot start before the end $t + p_k$ of the batch containing k so, we are exactly in the state (k, t, p_k, e, π_e) of the recursion. The other jobs must be completed before or in the batch containing J_k so, we are exactly in the state (k, s, π_s, t, p_k). Hence, we have $F(k, s, \pi_s, e, \pi_e) \geq F(k, s, \pi_s, t, p_k) + F(k, t, p_k, e, \pi_e)$.

We have proven that $F(k, s, \pi_s, e, \pi_e)$ is greater than or equal to the minimum of $F(k-1, s, \pi_s, e, \pi_e)+f_k(e+\pi_e)$ and of $\min_{t\in[s+\pi_s, e-p_k]} F(k, s, \pi_s, t, p_k)+F(k, t, p_k, e, \pi_e)$. Now we prove that $F(k, s, \pi_s, e, \pi_e)$ is lower than or equal to the minimum of the two expressions above.

- First consider a schedule that realizes $F(k-1, s, \pi_s, e, \pi_e)$. We can add k in the right fictive batch (the additional cost is $f_k(e+\pi_e)$) and we obtain a feasible schedule for the state (k, s, π_s, e, π_e). So, $F(k, s, \pi_s, e, \pi_e) \le F(k-1, s, \pi_s, e, \pi_e) + f_k(e+\pi_e)$.
- Now let t be the value that realizes $\min_{t\in[\max(s+\pi_s, r_k), e-p_k]} F(k, s, \pi_s, t, p_k) + F(k, t, p_k, e, \pi_e)$ and let us merge the schedules that respectively realize $F(k, s, \pi_s, t, p_k)$ and $F(k, t, p_k, e, \pi_e)$. The merging is possible because the right fictive batch of the first one coincides with the left fictive batch of the second one. This schedule is feasible for the state (k, s, π_s, e, π_e). So, $F(k, s, \pi_s, e, \pi_e) \le F(k, s, \pi_s, t, p_k) + F(k, t, p_k, e, \pi_e)$.

□

If $P = \sum p_i$ denotes the time horizon, the time complexity of the dynamic programming algorithm is $O(nP^5)$ since there are $O(nP^4)$ states and each time the parameter t can take at most P values.

Now consider the special case with identical processing times $\forall i, p_i = p$. First, note that the parameters π_s and π_e become un-useful. Moreover, it is easy to see that there are few relevant starting times since in active schedules, starting times are equal to a release date modulo p. Hence the set of relevant starting times is $O(n^2)$ and consequently, the complexity of the algorithm reduces to $O(n^{11})$.

5.2 Batching to Minimize an Ordered Objective Function

From now on, we restrict our study to ordered objective functions $\sum f_i$ (see Definition 2, Chapter 2). Hence, a kind of due-date δ_i is attached to each function f_i and we have $\delta_1 \le \cdots \le \delta_n$. By analogy with due date scheduling, we say that a job is late when it is completed after δ_i and that it is on-time otherwise. The "late" cost is ω_i and the early cost is time-dependant. Notice that a late job can be scheduled arbitrary late.

Several dominance properties are stated in §5.2.1. The dynamic programming algorithms for serial and parallel problems are described in §5.2.2 and §5.2.3. Finally in §5.2.4 we show that our approach can be extended to handle $T_{\max}$ and we draw some conclusions.

5.2.1 Dominance Properties

We first define two sets of time points at which batches start on active schedules. We then study a dominance related to ordered objective functions and finally we show that both dominances can be combined.

Starting Times

One of the reasons why it is easy to schedule equal length jobs is that there are few possible starting times. Indeed, active schedules are dominant and thus starting times are equal to a release date modulo p. This idea can be extended to batching problems.

Serial Batching

Ordered objective functions are regular, hence active schedules are dominant. Consequently, each batch starts either at the release date of one of its jobs or immediately after the completion time of another batch plus the setup time s. Hence we can assume that batches start and end in Θ', where

$$\Theta' = \{r_\lambda + \mu p + \nu s, \lambda \in \{1, \cdots, n\}, \mu \in \{0, \cdots, n\}, \nu \in \{0, \cdots, n\}\}$$

Parallel Batching

Because jobs have the same processing time p we can assume that the length of a batch is p. On top of that, active schedules are dominant so, batches start and end in Θ as defined in Proposition 3 ($\Theta = \{t | \exists r_i, \exists l \in \{0, \ldots, n\}, t = r_i + lp\}$).

Notice that $|\Theta'| = O(n^3)$ and $|\Theta| = O(n^2)$.

Ordered Objective Functions

The following proposition holds both for serial and parallel problems.

Proposition 33. *Any feasible schedule can be transformed in a "better" one where for any pair of on-time jobs $J_u, J_v (u < v)$, being executed in batches starting respectively at t_u and t_v, either $t_u \leq t_v$ or $t_v < r_u$.*

Proof. Sketch. Let (u, v) be the smallest vector, according to the lexicographical order, such that $t_u > t_v$ and $t_v \geq r_u$. The jobs J_u and J_v may be exchanged and the value of the objective function is not increased since it is an ordered objective function. Moreover, it is easy to see that the smallest vector (u', v') that does not satisfy the condition on the new schedule is such that $(u', v') > (u, v)$. Repeated exchanges can be used to replace any schedule by a schedule which satisfies the condition for all u, v (the vector (u, v) increases at each step, hence the number of exchanges is finite). □

Combining All Dominance Properties

Now consider an optimal and active schedule and apply the exchanges of Proposition 33. The resulting schedule is still optimal and the set of starting times is kept the same. Hence we can combine both dominance criteria, *i.e.*, schedules such that

- batches start and are completed either in Θ' for serial problems, or in Θ for parallel problems
- and such that for any pair of jobs $J_u, J_v (u < v)$, being executed in batches starting respectively at t_u and t_v, either $t_u \leq t_v$ or $t_v < r_u$,

are dominant. In the following such schedules will be referred to as **Serial-Dominant** or **Parallel-Dominant** schedules.

5.2.2 Dynamic Programming for the Serial Problem

The dynamic program relies on the notion of **partial** batch. Usually, once the starting time t and the completion time $t + p\nu$ of a batch is known, it is easy to compute that $(t + p\nu - t)/p = \nu$ jobs are in the batch (because the batch is full). In a partial batch, there may be some hole and we only know that **at most** $(t + p\nu - t)/p$ jobs are in the batch. The batch is therefore "partial" since some additional jobs could be *a priori* added. In the following, the maximal number of jobs in a partial batch is denoted by the symbol ν, while μ denotes the number of jobs actually scheduled.

Before defining the variables of the Dynamic Program, let us introduce the set of jobs $V_k(t_l, t_r)$. For any integer $k \leq n$ and for any time points $t_l \leq t_r$, $V_k(t_l, t_r)$ is the set of jobs whose index is lower than or equal to k and whose release date is in the interval $(t_l, t_r]$.

$$V_k(t_l, t_r) = \{J_i | i \leq k, r_i \in (t_l, t_r]\}$$

Variables of the Dynamic Program

The dynamic search is controlled by 5 parameters $k, t_l, t_r, \nu_l, \nu_r$ and μ_r. Each combination of these parameters defines a sub-problem involving the jobs in $V_k(t_l, t_r)$. As explained below, the objective of the sub-problem is to minimize $\sum_{J_i \in V_k(t_l, t_r)} f_i(C_i)$ under some constraints.

Definition 17. *A schedule $\mathcal{K}$ is Serial-Dominant for $(k, t_l, t_r, \nu_l, \nu_r, \mu_r)$ iff*

- *$\mathcal{K}$ is Serial-Dominant,*
- *jobs in $V_k(t_l, t_r)$ are either late or are completed before or at $t_r + p\nu_r$,*
- *a partial batch, containing 0 job of $V_k(t_l, t_r)$, starts at t_l and is completed at $t_l + p\nu_l$,*
- *a partial batch, containing μ_r jobs of $V_k(t_l, t_r)$, starts at t_r and is completed at $t_r + p\nu_r$.*

Now we can define the variables of the dynamic program.

Definition 18. *$S_k(t_l, t_r, \nu_l, \nu_r, \mu_r)$ is the minimal value taken by the function*

$$\sum_{J_i \in V_k(t_l, t_r)} f_i(C_i) \tag{5.1}$$

over the Serial-Dominant schedules for $(t_l, t_r, \nu_l, \nu_r, \mu_r)$. If there is no such schedule,

$$S_k(t_l, t_r, \nu_l, \nu_r, \mu_r) = +\infty.$$

Given this definition $S_0(t_l, t_r, \nu_l, \nu_r, \mu_r)$ equals 0 if $t_l + p\nu_l + s \leq t_r$ and $+\infty$ otherwise.

To get an intuitive picture of the decomposition scheme, assume that $J_k \in V_k(t_l, t_r)$ is on-time (if it is not, take into account the "late" cost $f_k(\delta_k)$ and set k to $k-1$) and consider the time point t at which the batch containing J_k starts on $\mathcal{K}$ (*cf.* Figure 5.1).

- If $t = t_r$, *i.e.*, if J_k is put in the right partial batch, then we take into account the cost $f_k(t_r + p\nu_r)$ associated to J_k and we solve the sub-problem defined by $(k-1, t_l, t_r, \nu_l, \nu_r, \mu_r - 1)$.
- If $t < t_r$, then J_k is scheduled in an in-between batch starting at t and containing ν jobs. Thanks to Proposition 33, on-time jobs of $U_{k-1}(t_l, t)$ are all completed before or at the completion time of this batch. On-time jobs of $U_{k-1}(t, t_r)$ are completed after this batch. Hence, we have a left sub-problem defined by $(k-1, t_l, t, \nu_l, \nu, \nu - 1)$ and a right one defined by $(k-1, t, t_r, \nu, \nu_r, \mu_r)$

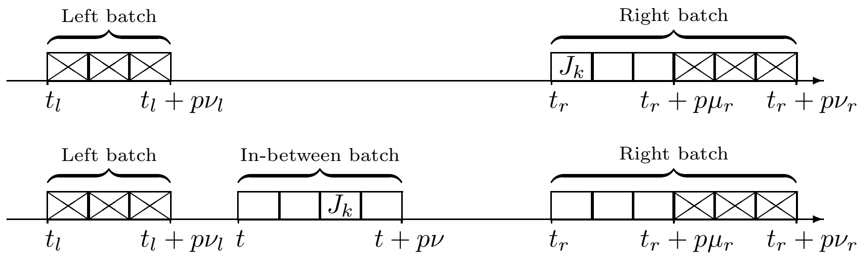


Figure 5.1: Two ways to schedule J_k: Either in the right partial batch or in an in-between batch (J_k can also be late). Ticked boxes in a batch indicate that it is only partially available for the jobs of $V_k(t_l, t_r)$.

Fundamental Recursion Formula

We define three values L, R, I. We will see that they are the costs of optimal schedules where J_k is in the **L**eft, in the **R**ight or in the **I**n-between batch.

Definition 19. *L equals $S_{k-1}(t_l, t_r, \nu_l, \nu_r, \mu_r) + f_k(\delta_k)$*

Definition 20. *R equals $f_k(t_r + p\nu_r) + S_{k-1}(t_l, t_r, \nu_l, \nu_r, \mu_r - 1)$ if $\mu_r > 0$ and $+\infty$ otherwise.*

Definition 21. *I is the minimum of*

$$S_{k-1}(t_l, t, \nu_l, \nu, \nu-1) + f_k(t+p\nu) + S_{k-1}(t, t_r, \nu, \nu_r, \mu_r) \tag{5.2}$$

under the constraints

$$\begin{cases} \nu \in \{1, \cdots, n\} \\ t \in \Theta' \\ r_k \leq t \leq \delta_k - p\nu \\ t_l + p\nu_l + s \leq t \leq t_r - p\nu - s \end{cases} \tag{5.3}$$

If there are no such t and ν, $I = +\infty$.

The following propositions lead to Proposition 38 that provides the fundamental recursion formula of the dynamic program.

Proposition 34. *If $J_k \in V_k(t_l, t_r)$ then $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r) \leq L$.*

Proof. Assume that L takes a finite value and let $\mathcal{R}$ be the schedule that realizes it. $\mathcal{R}$ is Serial-Dominant for $(k-1, t_l, t_r, \nu_l, \nu_r, \mu_r)$ and also for $(k, t_l, t_r, \nu_l, \nu_r, \mu_r)$ (J_k is late). The additional cost is exactly $f_k(\delta_k)$. □

Proposition 35. *If $J_k \in V_k(t_l, t_r)$ then $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r) \leq R$.*

Proof. Assume that R takes a finite value and let $\mathcal{Q}$ be the schedule that realizes $S_{k-1}(t_l, t_r, \nu_l, \nu_r, \mu_r - 1)$. Let $\mathcal{Q}'$ be the schedule obtained by adding the job J_k in the right batch (it can be added because it is released before the starting time of the batch since $J_k \in V_k(t_l, t_r) \Rightarrow r_k \leq t_r$). On the new schedule, at most $\mu_r - 1 + 1 = \mu_r$ jobs of $V_k(t_l, t_r)$ are scheduled in the right batch. It is easy to verify that $\mathcal{Q}'$ is a Serial-Dominant schedule for $(k, t_l, t_r, \nu_l, \nu_r, \mu_r)$. The additional cost is $f_k(t_r + p\nu_r)$. Hence $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r) \leq R$ □

Proposition 36. *If $J_k \in V_k(t_l, t_r)$ then $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r) \leq I$.*

Proof. Assume that I takes a finite value and let t and ν be the values that realize 5.2 and meet the constraints 5.3. Let $\mathcal{K}_l$ and $\mathcal{K}_r$ be the schedules that realize respectively $S_{k-1}(t_l, t, \nu_l, \nu, \nu-1)$ and $S_{k-1}(t, t_r, \nu, \nu_r, \mu_r)$. We build a schedule $\mathcal{K}'$ by adding the schedules $\mathcal{K}_l$ and $\mathcal{K}_r$ and by adding J_k in the right batch of $\mathcal{K}_l$. Since the set $J_k(t_r, t_l)$ is the direct sum of $J_{k-1}(t_r, t)$ plus $J_{k-1}(t, t_l)$ plus $\{J_k\}$, all jobs of $J_k(t_r, t_l)$ are scheduled exactly once on $\mathcal{K}'$. J_k can be added to the right batch because no more than $\nu - 1$ jobs are scheduled in this batch on $\mathcal{K}_l$. At least s time units elapse between batches (because $t_l + p\nu_l + s \leq t \leq t_r - p\nu - s$) and it is easy to verify that the in-between batch starts after the release dates of its jobs and before their due dates. We have proven that $\mathcal{K}'$ is feasible.
We claim that $\mathcal{K}'$ is a Serial-Dominant schedule for $(k, t_l, t_r, \nu_l, \nu_r, \mu_r)$. We only prove that "for any pair of jobs $J_u, J_v (u < v)$, being executed in batches starting respectively at t_u and t_v, either $t_u \leq t_v$ or $t_v < r_u$". The verification of all other conditions is easy and is left to the reader. If $v = k$ and if $t_v < t_u$ then J_u belongs to the sub-schedule $\mathcal{K}_r$ and thus, $J_u \in U_{k-1}(t, t_r)$, which leads to $t = t_v < r_u$ and the condition holds. Now assume that $v < k$. If both jobs belong to the same sub-schedule either $\mathcal{K}_r$ or $\mathcal{K}_l$, the condition holds because they are Serial-Dominant. Now assume that it is not the case and that $t_v < t_u$. We know that $t \in [t_v, t_u)$ and since $J_u \in U_{k-1}(t, t_r)$, $t_v \leq t < r_u$.
On $\mathcal{K}'$, the batch of J_k is completed at $t + p\nu$, hence the total cost of $\mathcal{K}'$ is $S_{k-1}(t_l, t, \nu_l, \nu, \nu-1) + f_k(t+p\nu) + S_{k-1}(t, t_r, \nu, \nu_r, \mu_r) = I$. As a consequence, $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r) \leq I$. □

Proposition 37. *If $J_k \in V_k(t_l, t_r)$ then $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r) \geq \min(I, R, L)$.*

Proof. We can assume that $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r)$ takes a finite value otherwise the proposition obviously holds. Let $\mathcal{W}$ be the schedule that realizes $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r)$. If J_k is not scheduled on $\mathcal{W}$, *i.e.*, it is late, then $\mathcal{W}$ is also Serial-Dominant for $(k-1, t_l, t_r, \nu_l, \nu_r, \mu_r)$ and the cost is decreased of $f_k(\delta_k)$ (because J_k is not taken into account). Hence, $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r) \geq L$. Now assume that J_k is scheduled on-time on $\mathcal{W}$ and let t be the time point at which the batch containing J_k starts and let ν be the size of this batch.

- If J_k is scheduled in the last batch, *i.e.*, if $t = t_l$, then remove it. It is easy to verify that the resulting schedule is Serial-Dominant for $(k-1, t_l, t_r, \nu_l, \nu_r, \mu_r - 1)$ and its cost is $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r) - f_k(t_r + p\nu_r)$. Hence, $S_{k-1}(t_l, t_r, \nu_l, \nu_r, \mu_r - 1) \leq S_k(t_l, t_r, \nu_l, \nu_r, \mu_r) - f_k(t_r + p\nu_r)$. Consequently, $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r) \geq R$.

- If J_k is not scheduled in the last batch then, because of the setup time constraint, it must start after or at $t_l + p\nu_l + s$ and it must be completed before or at $t_r - s$. Hence, $t \in [t_l + p\nu_l + s, t_r - p\nu - s]$. Moreover, the batch starts before the release date of J_k and is completed before its due date. Finally, the batch starts at a time point of Θ' because $\mathcal{W}$ is Serial-Dominant. Hence, constraints 5.3 are met. Let now $\mathcal{W}_l$ be the sub-schedule obtained from $\mathcal{W}$ by removing the jobs that are scheduled in batches starting strictly after t and by removing J_k. This sub-schedule is Serial-Dominant for $(k-1, t_l, t, \nu_l, \nu, \nu - 1)$ and thus its cost is greater than or equal to $S_{k-1}(t_l, t, \nu_l, \nu, 1)$. Similarly, let $\mathcal{W}_r$ be the sub-schedule obtained from $\mathcal{W}$ by removing the jobs that are scheduled in a batch starting before or at t. $\mathcal{W}_r$ is Serial-Dominant for $(k-1, t, t_r, \nu, \nu_r, \mu_r)$ and its cost is greater than or equal to $S_{k-1}(t, t_r, \nu, \nu_r, \mu_r)$. Finally, notice that the cost of $\mathcal{W}$, $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r)$, is the sum of the costs of $\mathcal{W}_l$ and $\mathcal{W}_r$ plus the cost of scheduling J_k in the batch starting at t, *i.e.*, $f_k(t + p\nu)$. Altogether, this leads to $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r) \geq I$.

Hence, $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r)$ is greater than or equal to either I, R or L. □

Proposition 38. *If $J_k \in V_k(t_l, t_r)$ then $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r) = \min(I, R, L)$. Otherwise,*

$$S_k(t_l, t_r, \nu_l, \nu_r, \mu_r) = S_{k-1}(t_l, t_r, \nu_l, \nu_r, \mu_r).$$

Proof. If $J_k \notin V_k(t_l, t_r)$ then $V_k(t_l, t_r) = U_{k-1}(t_l, t_r)$ and thus the proposition obviously holds. If $J_k \in V_k(t_l, t_r)$ then the result comes immediately from Propositions 35, 36, 34 and 37. □

An $O(n^{14})$ Algorithm

There is an optimal schedule that is serial-dominant and it comes directly from the definition of $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r)$ that the optimum is met for

$$\begin{cases} k & := & n \\ t_l & := & \min(\Theta') - s - 1 \\ \nu_l & := & 0 \\ t_r & := & \max(\Theta') + s \\ \nu_r & := & 0 \\ \mu_r & := & 0 \end{cases} \tag{5.4}$$

Notice that if $s > 0$ then t_l can be set to $\min(\Theta') - s$ instead of $\min(\Theta') - s - 1$ because $U_n(\min(\Theta') - s, \max(\Theta') + s) = \{J_1, \cdots, J_n\}$; which is not the case when $s = 0$.

Thanks to Proposition 38, we have a straight dynamic programming algorithm to reach the optimum. The relevant values for t_l and t_r are those in Θ' plus the special values $\min(\Theta') - s - 1$ and $\max(\Theta') + s$ that are useful to start the dynamic search (to simplify the pseudo-code, these special values have been omitted in Algorithm 5). The relevant values for k, ν_l, ν_r and μ_r are $\{0, \cdots, n\}$. Finally, the values of $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r)$ are stored in a multi-dimensional array of size $O(n^{10})$ (n possible values for k, ν_l, ν_r, μ_r, and n^3 possible values both for t_l and t_r).

In the initialization phase, $S_0(t_l, t_r, \nu_l, \nu_r, \mu_r)$ is set to 0 if $t_l + p\nu_l + s \leq t_r$ and to a very large value otherwise. The initialization phase runs in $O(n^9)$ (n possible values for ν_l, ν_r, μ_r, n^3 possible values both for t_l and t_r).

We then iterate on all possible values of the parameters to reach the optimum (*cf.* Algorithm 5). Each time, a minimum over $O(n^4)$ terms ($O(n^3)$ for t and $O(n)$ for ν) is computed. This leads to an overall time complexity of $O(n^{14})$. A rough analysis of the space complexity leads to an $O(n^{10})$ bound but since, at each step of the outer loop on k, one only needs the values of S computed at the previous step $(k-1)$, the algorithm can be implemented with 2 arrays of $O(n^9)$ size: one for the current values of S and one for the previous values of S. (To build the optimal schedule, all values of S have to be kept; hence the initial $O(n^{10})$ bound holds.)

Algorithm 5 Computation of the values $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r)$

```
for $k := 1$ to $n$ do
  for $\nu_r := 0$ to $n$ do
    for $\nu_l := 0$ to $n$ do
      for $\mu_r := 0$ to $\nu_r$ do
        for $t_r \in \Theta'$ taken in increasing order do
          for $t_l \in \Theta'$ ($t_l \leq t_r$) taken in increasing order do
            $R := +\infty, I := +\infty, L := +\infty$
            if $\mu_r > 0$ then
              $R := f_k(t_r + p\nu_r) + S_{k-1}(t_l, t_r, \nu_l, \nu_r, \mu_r - 1)$
            end if
            for $\nu := 1$ to $n$ do
              for $t \in \Theta' \cap [t_l + p\nu_l + s, t_r - p\nu - s] \cap [r_k, \delta_k - p\nu]$ do
                $I := \min(I, S_{k-1}(t_l, t, \nu_l, \nu, \nu - 1) + f_k(t + p\nu) + S_{k-1}(t, t_r, \nu, \nu_r, \mu_r))$
              end for
            end for
            $L := S_{k-1}(t_l, t_r, \nu_l, \nu_r, \mu_r) + f_k(\delta_k)$
            $S_k(t_l, t_r, \nu_l, \nu_r, \mu_r) := \min(I, R, L)$
          end for
        end for
      end for
    end for
  end for
end for
```

5.2.3 Dynamic Programming for the Parallel Problem

Variables of the Dynamic Program

The dynamic search is controlled by 4 parameters k, t_l, t_r and μ_r. Each combination of these parameters defines a sub-problem involving the jobs in $V_k(t_l, t_r)$. The objective is to minimize $\sum_{J_i \in V_k(t_l,t_r)} f_i(C_i)$ under some constraints.

Definition 22. *A schedule $\mathcal{K}$ is Parallel-Dominant for (k, t_l, t_r, μ_r) iff*

- *$\mathcal{K}$ is Parallel-Dominant,*
- *jobs in $V_k(t_l, t_r)$ do not start before $t_l + p$,*
- *jobs in $V_k(t_l, t_r)$ are either late or are completed before or at $t_r + p$,*
- *a partial batch, containing μ_r jobs of $V_k(t_l, t_r)$, starts at t_r.*

Now we can define the variables of the dynamic program.

Definition 23. *$P_k(t_l, t_r, \mu_r)$ is the minimal value taken by the function*

$$\sum_{J_i \in V_k(t_l,t_r)} f_i(C_i) \tag{5.5}$$

over the Parallel-Dominant schedules of $V_k(t_l, t_r)$. If there is no such schedule,

$$P_k(t_l, t_r, \mu_r) = +\infty.$$

Given this definition $P_0(t_l, t_r, \mu_r)$ equals 0 if $t_l + p \leq t_r$ and $+\infty$ otherwise.

To get an intuitive picture of the decomposition scheme, assume that $J_k \in V_k(t_l, t_r)$ is on-time (if it is late, k can be decreased of 1 and we only have to take into account the cost of scheduling J_k late, *i.e.*, $f_k(\delta_k)$) and consider the time point t at which the batch containing J_k starts on $\mathcal{K}$ (*cf.* Figure 5.2).

- If $t = t_r$, *i.e.*, if J_k is put in the right batch, we have to solve the same problem with $k := k - 1, t_l := t_l, t_r := t_r$ and $\mu_r := \mu_r - 1$.
- If $t < t_r$, then an in-between batch is created. Thanks to Proposition 33 jobs of $U_{k-1}(t_l, t)$ are either late or are completed before or at the completion time of this batch. Jobs in $U_{k-1}(t, t_r)$ are completed after this batch. Hence, we have two sub-problems. A left one with $(k - 1, t_l, t, b - 1)$ and a right one with $(k - 1, t, t_r, \mu_r)$.

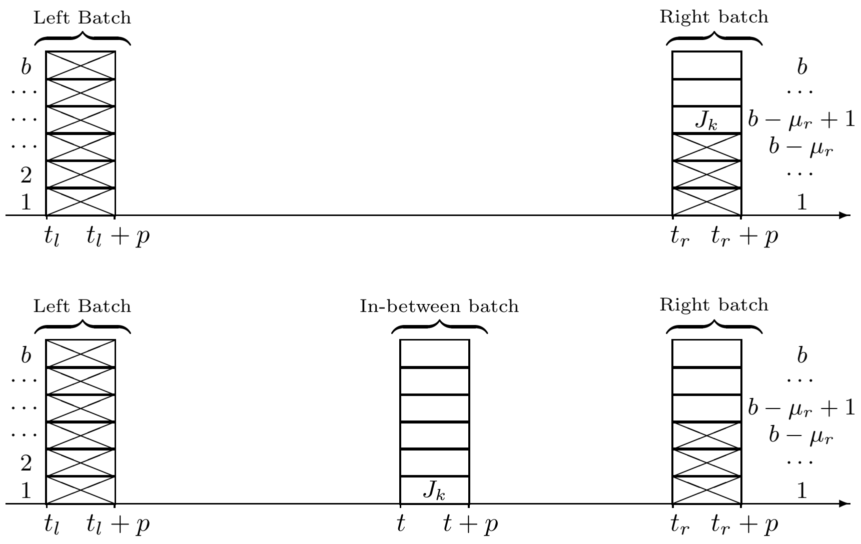


Figure 5.2: Two ways to schedule J_k: Either in the right batch or in an in-between batch (J_k can also be late). Ticked boxes in a batch indicate that it is only partially available for the jobs of $V_k(t_l, t_r)$.

Fundamental Recursion Formula

Let us redefine the values of R, I, L.

Definition 24. *L equals $P_{k-1}(t_l, t_r, \mu_r) + f_k(\delta_k)$*

Definition 25. *R equals $f_k(t_r + p) + P_{k-1}(t_l, t_r, \mu_r - 1)$ if $\mu_r > 0$ and $+\infty$ otherwise.*

Definition 26. *I is the minimum of*

$$P_{k-1}(t_l, t, b - 1) + f_k(t + p) + P_{k-1}(t, t_r, \mu_r) \tag{5.6}$$

under the constraints

$$\begin{cases} t \in \Theta \\ r_k \leq t \leq \delta_k - p \\ t_l + p \leq t \leq t_r - p \end{cases} \tag{5.7}$$

If there is no such t, $I = +\infty$.

We are ready to formulate the fundamental proposition.

Proposition 39. *If $J_k \in V_k(t_l, t_r)$ then $P_k(t_l, t_r, \mu_r) = \min(I, R, L)$. Otherwise,*

$$P_k(t_l, t_r, \mu_r) = P_{k-1}(t_l, t_r, \mu_r).$$

Proof. See proof of Proposition 38. □

Algorithm 6 Computation of the values $P_k(t_l, t_r, \mu_r)$

```
for $k := 1$ to $n$ do
  for $\mu_r := 0$ to $b$ do
    for $t_r \in \Theta$ taken in increasing order do
      for $t_l \in \Theta$ $(t_l \le t_r)$ taken in increasing order do
        $R := +\infty, I := +\infty, L := +\infty$
        if $\mu_r > 0$ then
          $R := f_k(t_r + p) + P_{k-1}(t_l, t_r, \mu_r - 1)$
        end if
        for $t \in \Theta \cap [t_l + p, t_r - p] \cap [r_k, \delta_k - p]$ do
          $I := \min(I, P_{k-1}(t_l, t, b-1) + f_k(t+p) + P_{k-1}(t, t_r, \mu_r))$
        end for
        $L := P_{k-1}(t_l, t_r, \mu_r) + f_k(\delta_k)$
        $P_k(t_l, t_r, \mu_r) := \min(I, R, L)$
      end for
    end for
  end for
end for
```

An $O(n^8)$ Algorithm

There is an optimal Parallel-Dominant schedule and it comes directly from the definition of $P_k(t_l, t_r, \mu_r)$ that the optimum is met for

$$\begin{cases} k & := & n \\ t_l & := & \min(\Theta) - p \\ t_r & := & \max(\Theta) + p \\ \mu_r & := & 0 \end{cases} \tag{5.8}$$

Thanks to Proposition 39, we have a straight dynamic programming algorithm to reach the optimum. The relevant values for t_l and t_r are those in Θ plus the special values $\min(\Theta) - p$ and $\max(\Theta) + p$. The relevant values for k and μ_r are in $\{0, \cdots, n\}$. The values of $P_k(t_l, t_r, \mu_r)$ are stored in a multi-dimensional array of size $O(n^6)$.

In the initialization phase, $P_0(t_l, t_r, \mu_r)$ is set to 0 if $t_l + p \le t_r$ and to a very large value otherwise. It runs in $O(n^5)$. We then iterate on all possible values of the parameters to reach the optimum (*cf.* Algorithm 6). For each value of the parameters, a minimum over t ($O(n^2)$ terms) is computed. This leads to an overall time complexity of $O(n^8)$. A rough analysis of the space complexity leads to an $O(n^6)$ bound but, as for the serial problem, the algorithm can be implemented with 2 arrays of $O(n^5)$ size.

Bar-Noy, Guha, Katz, Naor, Schieber and Shachnai [28] have proposed an $O(n^{F^2+3F+2} \log n)$ dynamic programming algorithm for the more general case where F families of jobs, all jobs in the same familiy having identical processing times, have to be batched on a parallel batching machine. This improves on the result presented above.

5.2.4 Open Questions

We have shown that with arbitrary release dates and identical processing times, (1) the unbounded parallel problem is solvable in polynomial time for any regular objective function, (2) the parallel and the serial problems are also solvable in polynomial time for any ordered objective function. As shown in Chapter 2, $\sum w_i U_i$, $\sum w_i C_i$ or $\sum T_i$ are ordered objective functions or can be transformed into ordered objective functions. Hence, we have shown that the problems

- $1|s\text{-}batch, r_i, p_i = p|\sum w_i U_i$,
- $1|s\text{-}batch, r_i, p_i = p|\sum w_i C_i$,

- $1|s\text{-}batch, r_i, p_i = p|\sum T_i$,
- $1|p\text{-}batch, b < n, r_i, p_i = p|\sum w_iU_i$,
- $1|p\text{-}batch, b < n, r_i, p_i = p|\sum w_iC_i$,
- $1|p\text{-}batch, b < n, r_i, p_i = p|\sum T_i$,
- $1|p\text{-}batch, r_i, p_i = p|\sum w_iT_i$,

are solvable in polynomial time by dynamic programming. To conclude we would like to point out that, for the criteria $\sum w_iT_i$, the complexity of batching identical jobs on a serial or on a parallel bounded machine is still an open question.

Chapter 6

Conclusion

In this first part, we have provided a set of techniques to solve a large variety of equal–processing–time scheduling problems. Most of the problems studied were open before and we believe that our algorithms contribute to a better understanding, as far as scheduling is concerned, of the borderline between NP-Hard and polynomial problems. Although we have tried to be very exhaustive, there are still many “simple” open equal–execution time scheduling problems.

When there are no precedence constraints, a very challenging problem is $1|p_i = p, r_i|\sum w_i T_i$. It seems that our dynamic programming techniques cannot be extended to this case. Indeed, to decompose the problem, we always try to find the “less urgent” job in the instance. It is easy to see that for $\sum w_i C_i$, the less urgent job is the one with minimal weight and for $\sum T_i$, it is the one with largest due-date. The main issue with the weighted tardiness criteria is that a job with a very large due-date can also have a very large weight. Hence, it seems difficult to generalize this notion of “less urgent” job.

Many preemptive equal–execution–time problems are also open according to the Brucker and Knust complexity list [47]. Although the corresponding results are not described in this manuscript, we have shown recently that, for several problems, preemption is redundant. This allows us to close a preemptive problem when there is a polynomial solution of its non-preemptive counterpart. Together with Timkovsky [26] we have shown that preemption is redundant for $P2|pmtn, outtree, r_j, p_j = 1|\sum C_j$ (and thus the problem can be solved in polynomial time with Hu’s algorithm). Brucker, Hurink and Knust have extended this result to an arbitrary number of machines [42]. At the same time, we have shown with Timkovsky that preemption is also redundant for an arbitrary precedence graph on a single machine ($1|pmtn, prec, r_j, p_j = p|\sum C_j$). More recently, Brucker, Heitmann and Hurink [40] have shown that preemption is redundant for $P|pmtn, p_j = 1|\sum T_j$. We believe that all these existing results (and more) can be presented in a common generic framework. Preemption redundancy is, we think, a very promising research direction.

Finally, we would like to mention some negative results on the parallel unit execution time problems with precedence constraints. $P3|prec|C_{\max}$, one of the most challenging problem in the history of scheduling, is still open. Despite our efforts, it seems that none of the techniques described in this manuscript can be, *a priori* generalized to handle arbitrary precedence constraints.

Part II

Constraint-Based Scheduling

Chapter 7

Introduction

Constraint-Based Scheduling can be defined as the discipline that studies how to solve scheduling problems by using Constraint Programming (CP). In this introduction we first pay attention to the basics of CP, after which we introduce the scheduling problems that are considered.

7.1 Constraint Programming

Constraint Programming is a paradigm aimed at solving combinatorial optimization problems. Often these combinatorial optimization problems are solved by defining them as one or several instances of the *Constraint Satisfaction Problem* (CSP). Informally speaking, an instance of the CSP is described by a set of *variables*, a set of possible values for each variable, and a set of *constraints* between the variables. The set of possible values of a variable is called the variable's *domain*. A constraint between variables expresses which combinations of values for the variables are allowed. Constraints can be stated either implicitly (also called intentionally), *e.g.*, an arithmetic formula, or explicitly (also called extensionally), where each constraint is expressed as a set of tuples of values that satisfy the constraint. An example of an implicitly stated constraint on the integer variables x and y is $x < y$. An example of an explicitly stated constraint on the integer variables x and y with domains $\{1, 2, 3\}$ and $\{1, 2, 3, 4\}$ is the tuple set $\{(1, 1), (2, 3), (3, 4)\}$. The question to be answered for an instance of the CSP is whether there exists an assignment of values to variables, such that all constraints are satisfied. Such an assignment is called a *solution* of the CSP.

One of the key ideas of CP is that constraints can be used "actively" to reduce the computational effort needed to solve combinatorial problems. Constraints are thus not only used to test the validity of a solution, as in conventional programming languages, but also in an active mode to remove values from the domains, deduce new constraints, and detect inconsistencies. This process of actively using constraints to come to certain deductions is called *constraint propagation*. The specific deductions that result in the removal of values from the domains are called *domain reductions*. The set of values in the domain of a variable that are not invalidated by constraint propagation is called the *current domain* of that variable. As an example of the benefit of the active use of constraints, let us look at the combination of the three constraints $y > x$, $x > 8$, and $y \leq 9$ on the integer variables x and y. Looking at the first two constraints, we can deduce that the value of y is greater than 9. This is administered in the current domain of y. Then by using the constraint $y \leq 9$, obviously a contradiction can be detected. Without constraint propagation, the "$y \leq 9$" test could not be performed before the instantiation of y and thus no contradiction would be detected. The current domains of the variables play a central role in constraint propagation as a basic means of communication between constraints. In this small example the fact the current domain of y only includes values greater than 9 makes it trivial for the constraint $y \leq 9$ to detect a contradiction.

As the general CSP is NP-complete [108], constraint propagation is usually incomplete. This means that some but not all the consequences of the set of constraints are deduced. In particular, constraint propagation cannot detect all inconsistencies. Consequently, one needs to perform some kind of search to determine if the CSP instance at hand has a solution or not. Most commonly, search is performed by means of a tree search algorithm. The two main components of a tree search algorithm are (i) the way to go "forward", *i.e.*, the definition of which decisions are taken at which point in the search, and (ii) the way to go "backward", *i.e.*, the definition of the backtracking strategy which states how the algorithm shall behave when a contradiction is detected. The description of which decisions to take at which point in the search is often referred to as the *search heuristic*. In general, the decisions that are taken correspond to adding additional constraints. As such during search the constraint propagation reasons on the combination of the original constraints and the constraints coming from the decisions taken. When a contradiction is detected it thus means it is proven that there is no feasible assignment of values to variables given the original data of the CSP and the heuristic decisions that have been made. The most commonly used backtracking strategy is depth first chronological backtracking, *i.e.*, the last decision is undone and an alternative constraint is imposed. More complex backtracking strategies can be found for example in [140, 52, 188]. In Section 12.3.4 we present an application where depth first chronological backtracking is compared to alternatives such as limited discrepancy search [116].

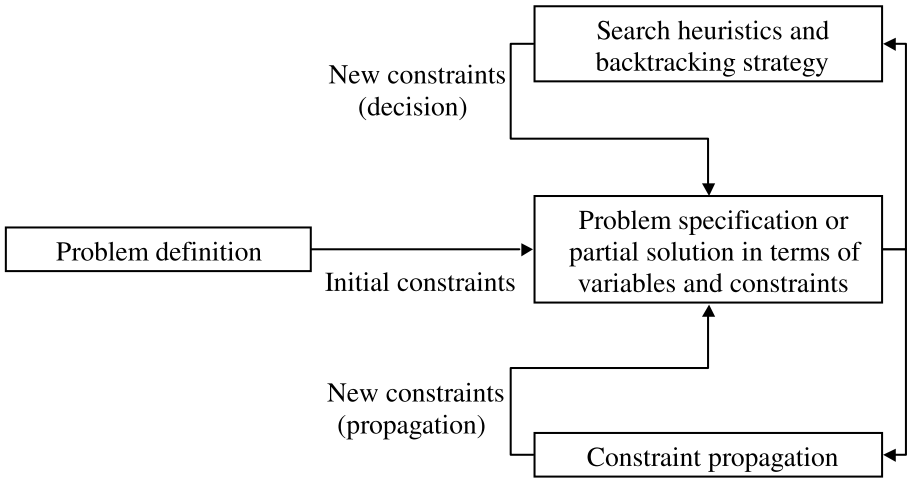

Search heuristics and backtracking strategy
New constraints (decision)
Problem definition
Initial constraints
Problem specification or partial solution in terms of variables and constraints
New constraints (propagation)
Constraint propagation

Figure 7.1: The behavior of a Constraint Programming system.

The overall behavior of a CP system is depicted in Figure 7.1. This figure underlines the fact that *problem definition*, *constraint propagation*, and the *search heuristic* and *backtracking strategy* are clearly separated. This separation is based upon the following fundamental principles of CP that became precise in the late 1970's and early 1980's. In [206] the separation between deductive methods generating additional constraints from existing constraints, and search algorithms used to systematically explore the solution space is advocated. For CP the deductive method is constraint propagation. The distinction between the logical representation of constraints and the control of their use is in accordance with the equation stated by Kowalski for logic programming: Algorithm = Logic + Control [132]. Another important principle used in CP is the so-called "locality principle", which states that each constraint must propagate as locally as possible, independently of the existence or the non-existence of other constraints [207].

When defining the problem in terms of variables and constraints, in practice the user of a CP tool is offered an array of pre-defined constraints (*e.g.*, constraints on integers, constraints on sets, scheduling constraints) with corresponding propagation algorithms. The locality principle is of crucial importance for the practical use of the pre-defined constraints as it implies that the constraint propagation algorithms can be reused in all applications where similar constraints apply. On top of this many CP tools offer ways to define new constraints with corresponding propagation algorithms that then seamlessly can be used in cooperation with the pre-defined constraints. CP tools also offer support to specify the search heuristic and the backtracking strategy. Again pre-defined heuristics are offered (*e.g.*, instantiate all variables by choosing the variable with the smallest current domain and assigning it the minimal value in that domain, order all activities on all resources) as well as ways to define one's own heuristics. The same goes for backtracking strategies: some pre-defined strategies are offered (*e.g.*, depth first chronological backtracking, limited discrepancy search) as well as ways to define one's own backtracking strategy.

All these properties together contributed to the success of commercial and public domain CP tools such as ILOG SOLVER [189, 190], CHIP [117, 4, 31], PROLOG III, IV [77], ECLIPSE [215], CLAIRE [69] and CHOCO [139]. For a comparison between several CP languages we refer to [101]. For an overview of CP, its principles, and its applications, we refer to [141, 124, 125, 134, 212, 99, 66, 165, 162, 36, 118].

To give an example of several of the aspects of solving a CSP mentioned above, we turn to one of the most popular examples in CP literature, namely the n-queens problem. The n-queens problem involves placing n queens on a chess board in such a way that none of them can capture any other using the conventional moves allowed to a queen. In other words, the problem is to select n squares on a $n \times n$ chess board so that any pair of selected squares is never aligned vertically, horizontally, nor diagonally. The problem is of limited practical importance, but it does allow to discuss subjects like modeling, constraint propagation, search, and backtracking in more detail.

Let's first look at the modeling of the problem. The n-queens problem can be modeled by introducing n integer variables x_i, each representing the position of the queen in the i-th row, that is, the column number where the i-th queen is located. As such the domain for each variable is the set of integers 1 to n. The constraints of the problem can be stated in the following way. For every pair (i, j), where i is different from j, $x_i \neq x_j$ guarantees that the columns are distinct and $x_i + i \neq x_j + j$ and $x_i - i \neq x_j - j$ together guarantee that the diagonals are distinct.

In Figure 7.2 we pay attention to solving the 6-queens problem. Each row in the larger squares correspond to a variable, the first row corresponding to x_1, *etc.* The columns of the squares correspond to the possible values for the variables, the first column corresponding to the value 1, *etc.* To keep the example simple, the search heuristic is defined to consider the variables x_1 to x_6 in that order and to assign them the minimal value in their current domain. Following the definition of the search algorithm, x_1 is assigned value 1 in Step 1, which is reflected by coloring the corresponding small square black. The values that were removed from the domains of the variables by constraint propagation due to this decision are colored gray. The current domain of a variable is thus represented by the white squares in its row. The constraint propagation prevents for instance that x_2 is assigned the value 1 or 2, only to find that this violates the constraints. This is an example of constraint propagation reducing the search effort.

Step 2 and 3 simply assign the smallest values in the current domain of x_2 and x_3 to these variables. In Step 3 (cont.) two effects take place. The constraint propagation leaves only one value for x_4, which for the propagation is equivalent to assigning this value to that variable. The propagation thus continues and finds that the current domain of x_6 gets empty and as such deduces that the original constraints plus the decisions are inconsistent. This deduction is made before all variables are assigned a value, again an example how search effort is reduced by constraint propagation.

In this example we simply employ depth first chronological backtracking. We thus go back to the last decision taken, in Step 3, and try an alternative choice, in Step 3'. Constraint propagation again deduces a contradiction, and as no alternatives for Step 3 exists, we backtrack to Step 2, where an alternative is tried, *etc.*, until we find a solution.

Although this problem is of little practical importance it does show that the combination of the model, the search heuristic, the constraint propagation, and the backtracking strategy defines the performance of a CP algorithm. As such if one wants to improve the performance of a CP approach, one needs to work on at least one of these components. We remark that in CP improving the model often is aimed at being able to define a better search heuristic or to have better (pre-defined) constraint propagation.

As already mentioned a crucial component in a CP approach is the constraint propagation. Several techniques have been developed to do constraint propagation. Among these techniques, let

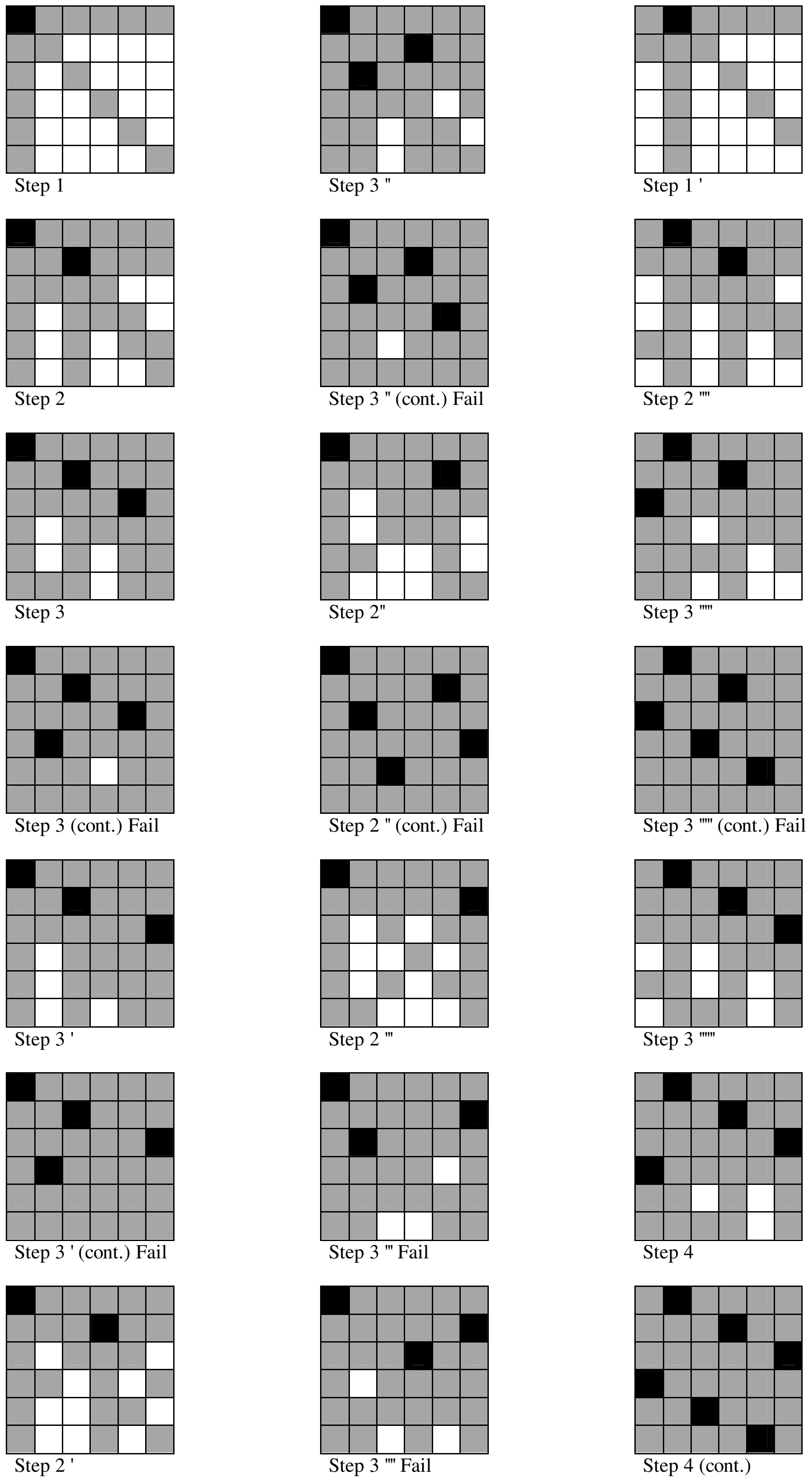


Figure 7.2: Solving the 6-queens problem.

us mention arc-consistency.

Definition 27. *Given a constraint c over n variables $x_1, \ldots, x_n$ and a domain $d(x_i)$ for each variable x_i, c is said to be "arc-consistent" if and only if for any variable x_i and any value v_i in $d(x_i)$, there exist values $v_1, \ldots, v_{i-1}$, $v_{i+1}, \ldots, v_n$ in $d(x_1), \ldots, d(x_{i-1})$, $d(x_{i+1}), \ldots, d(x_n)$ such that $c(v_1, \ldots, v_n)$ holds.*

A huge amount of work has been carried out on constraint propagation algorithms that maintain arc-consistency on the constraints of a binary CSP, *i.e.*, of a CSP whose constraints link at most two variables [169, 161, 167, 119, 33]. Numeric CSPs are special cases of the CSP where the variables are constrained to take numeric values. For numeric CSPs the domain of a variable x often is represented by only its smallest value $lb(x)$ and its largest value $ub(x)$, *i.e.*, by the interval $[lb(x), ub(x)]$. This compact representation is especially useful to tackle real-life problems for which explicitly maintaining the set of values that can be taken by each variable throughout the search tree may not be reasonable. A usual way to propagate constraints on such variables is to achieve arc-B-consistency [155], *i.e.*, arc-consistency restricted to the bounds of the domains. Arc-B-consistency can be easily achieved on some arithmetic constraints such as linear constraints [155].

Definition 28. *Given a constraint c over n variables $x_1, \ldots, x_n$ and a domain $d(x_i) = [lb(x_i), ub(x_i)]$ for each variable x_i, c is said to be "arc-B-consistent" if and only if for any variable x_i and each of the bound values $v_i = lb(x_i)$ and $v_i = ub(x_i)$, there exist values $v_1, \ldots, v_{i-1}$, $v_{i+1}, \ldots$, v_n in $d(x_1), \ldots, d(x_{i-1})$, $d(x_{i+1})$, $\ldots, d(x_n)$ such that $c(v_1, \ldots, v_n)$ holds.*

In the following, maintaining arc-consistency (or arc-B-consistency) on a constraint means that the domains (respectively the bounds) of the variables involved in the constraint are maintained to ensure that the constraint is arc-consistent (resp. arc-B-consistent).

Example 1. *Maintaining arc-consistency or arc-B-consistency on*

$$\sum_{i=1}^{n} \lambda_i x_i \leq \Lambda \tag{7.1}$$

where $\lambda_1, \ldots, \lambda_n$ and Λ are non negative values. It is easy to see that (7.1) leads to:

$$\forall i, x_i \leq \frac{\Lambda - \sum_{j \neq i} \lambda_j x_j}{\lambda_i}$$

Consequently, the upper bound of x_i can be adjusted to

$$\frac{\Lambda - \sum_{j \neq i} \lambda_j lb(x_j)}{\lambda_i}$$

This upper bound needs to be recomputed only when the lower bound of one of the variables $x_1, \ldots, x_{i-1}$, $x_{i+1} \ldots, x_n$ is modified.

Example 2. *Maintaining arc-consistency or arc-B-consistency on the disjunctive constraint $[x_1 = 0] \vee [x_2 = 0]$. As soon as the value 0 is removed from the domain of x_1 (resp. of x_2) then the second (resp. first) disjunct is imposed, i.e., $x_2 = 0$ (resp. $x_1 = 0$).*

Example 3. *Maintainig arc-consistency on $x_1 \neq x_2$. When x_1 is bound to v_1, this value is removed from the domain of x_2 (and conversely). Notice that in this example, arc-B-consistency and arc-consistency differ. Indeed, when only arc-B-consistency is maintained, v_1 is removed from $d(x_2)$ only if it is equal to either $lb(x_2)$ or $ub(x_2)$.*

In the past few years much evidence was gathered that the use of "global" constraint propagation algorithms can drastically enhance the efficiency of CP systems. Such algorithms are able to take into account a set of constraints from a global point of view, and can propagate them very efficiently.

Let us consider for instance the so-called "all-different" constraint. It constrains a set of n variables to take pairwise distinct values. Such a constraint can obviously be propagated by maintaining arc-consistency on $n(n-1)/2$ "local" constraints that state for any pair of variables x and y that $x \neq y$. An algorithm to achieve the consistency of the global "all-different" constraint is described in [191]. The constraint is modeled by a bi-partite graph. One of the sets of vertices is the set of the variables while the other one is the union of the domains. An edge between a variable and a value states that the given value is in the domain of the given variable. The constraint is consistent if and only if there is a matching whose cardinality is n. An algorithm is devised to ensure the arc-consistency of the global constraint. The domains of the variables are filtered to remove all the values that would make the constraint inconsistent. Another famous global constraint is the resource constraint that states that a set of activities cannot overlap in time (this constraint will be presented in detail in Chapter 8).

One of the key factors of the recent successes of Constraint-Based Scheduling lies in the fact that a combination was found of the best of two fields of research that pay attention to scheduling, namely *Operations Research* (OR) and *Artificial Intelligence* (AI).

Traditionally, a lot of the attention in OR has been paid to rather "pure" scheduling problems that are based on relatively simple mathematical models (*cf.*, First Part of this manuscript). For solving the problem at hand, the combinatorial structure of the problem is heavily exploited, leading to improved performance characteristics. We could say that an OR approach often aims at achieving a high level of *efficiency* in its algorithms. However, when modeling a practical scheduling problem using these classical models, one is often forced to discard degrees of freedom and side constraints that exist in the practical scheduling situation. Discarding degrees of freedom may result in the elimination of interesting solutions, regardless of the solution method used. Discarding side constraints gives a simplified problem and solving this simplified problem may result in impractical solutions for the original problem. In contrast, AI research tends to investigate more general scheduling models and tries to solve the problems by using general problem-solving paradigms. We could say an AI approach tends to focus more on the *generality of application* of its algorithms. This, however, implies that AI algorithms may perform poorly on specific cases, compared to OR algorithms.

So, on the one hand we have OR which offers us *efficient* algorithms to solve problems that in comparison have a more limited application area. On the other hand we have AI that offers us algorithms that are more *generally applicable*, but that might suffer from somewhat poor performance in the specific cases an efficient OR algorithm exists. An important way to combine the two was found by incorporating OR algorithms inside global constraints. Examples thereof are the already mentioned all-different constraint and the resource constraint on a single machine. The basics of many of the algorithms inside global constraints, certainly in the early stages of the field, can be found in OR. By applying the locality principle, such specialized algorithms can work side by side with general propagation algorithms that take care of the rest of the constraints, the side constraints, *etc.* In this way one can preserve the general modeling and problem-solving paradigm of CP while the integration of efficient propagation algorithms improves the overall performance of the approach. Stated in another way, efficient OR algorithms integrated in a CP tool allow the user to benefit from the efficiency of OR techniques in a flexible framework. All this said, we want to remark that over the years the distinction between AI and OR is often becoming less and less clear and is also deemed less and less important.

7.2 A Constraint-Based Scheduling Model

When looking at the type of resources found in a problem, we distinguish *disjunctive scheduling* and *cumulative scheduling.* In a disjunctive scheduling problem, all resources are machines and thus can execute at most one activity at a time. In a cumulative scheduling problem, resources exist that can execute several activities in parallel, of course provided that the resource capacity is not exceeded.

When looking at the type of activities in a problem, we distinguish *non-preemptive scheduling*, *preemptive scheduling*, and *elastic scheduling.* In non-preemptive scheduling, activities cannot be interrupted. Each activity must execute without interruption from its start time to its end time. In preemptive scheduling, activities can be interrupted at any time, *e.g.*, to let some other activities execute. In elastic scheduling the amount of resource assigned to an activity A_i can, at any time

t, assume any value between 0 and the resource capacity, provided that the sum over time of the assigned capacity equals a given value called *energy*. The equivalent notion of energy in the case of a non-preemptive activity is the product of its processing time and the capacity required

We furthermore distinguish *decision* problems and *optimization* problems. In decision problems, one has only to determine whether a schedule exists that meets all constraints. In optimization problems, an objective function has to be minimized. Although the minimization of the makespan, *i.e.*, the end time of the schedule, is commonly used, other criteria are sometimes of great practical interest (*e.g.*, the number of activities performed with given delays, the maximal or average tardiness or earliness, the peak or average resource utilization, the sum of setup times or costs).

7.2.1 Activities

A non-preemptive scheduling problem can be efficiently encoded as a CSP in the following way. For each activity three variables are introduced, $start(A_i)$, $end(A_i)$, and $proc(A_i)$. They represent the start time, the end time, and the processing time of A_i, respectively.

With r_i the release date and $\bar{d}_i$ the deadline of activity A_i as defined in the initial data of the scheduling problem, $[r_i, \bar{d}_i]$ is the time window in which A_i has to execute. Based on that the initial domains of $start(A_i)$ and $end(A_i)$ are $[r_i, lst_i]$ and $[eet_i, \bar{d}_i]$, respectively. Here lst_i and eet_i stand for the latest start time and the earliest end time of A_i. For convenience, we also use this notation to denote the current domains of $start(A_i)$ and $end(A_i)$, *i.e.*, the domains when we are in the process of propagating constraints. Of course in that case instead of the initial release date and deadline, r_i and $\bar{d}_i$ denote the current earliest start time and latest end time

The processing time of the activity is defined as the difference between the end time and the start time of the activity: $proc(A_i) = end(A_i) - start(A_i)$. p_i denotes the smallest value in the domain of $proc(A_i)$. All data related to an activity are summarized in Figure 7.3. Light gray is used to depict the time-window $[r_i, \bar{d}_i]$ of an activity and dark gray is used to represent the processing time of the activity.

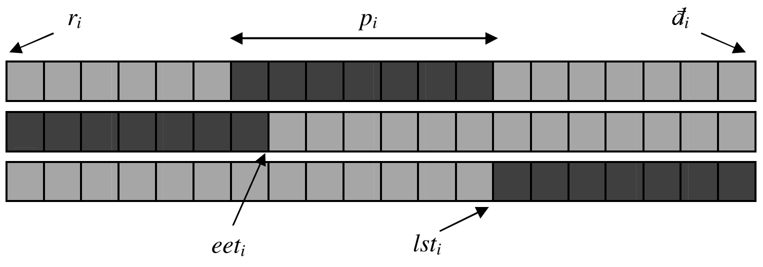


Figure 7.3: Data related to an activity

Preemptive scheduling problems are more difficult to represent since a schedule is more complex than a simple set of starting times of activities. We discuss two possibilities. One can either associate a set variable (*i.e.*, a variable the value of which will be a set) $set(A_i)$ with each activity A_i, or alternatively define a 0-1 variable $X(A_i, t)$ for each activity A_i and time t. $set(A_i)$ represents the set of times at which A_i executes, while $X(A_i, t)$ takes value 1 if and only if A_i executes at time t. The processing time $proc(A_i)$ of A_i is defined as the number of time points t at which A_i executes, *i.e.*, as $|set(A_i)|$. In practice, the $X(A_i, t)$ variables are not represented explicitly as the value of $X(A_i, t)$ is 1 if and only if t belongs to $set(A_i)$.

Assuming time is discretized, $start(A_i)$ and $end(A_i)$ can be defined by $start(A_i) = \min_{t \in set(A_i)} t$ and $end(A_i) = \max_{t \in set(A_i)} t{+}1$. Notice that in the non-preemptive case, $set(A_i) = [start(A_i), end(A_i))$, with the interval $[start(A_i), end(A_i))$ closed on the left and open on the right so that $|set(A_i)| = end(A_i) - start(A_i) = proc(A_i)$.

These constraints are easily propagated by maintaining a lower bound and an upper bound for the set variable $set(A_i)$. The lower bound $lb(set(A_i))$ is a series of disjoint intervals ILB_i^u such that each ILB_i^u is constrained to be included in $set(A_i)$. The upper bound $ub(set(A_i))$ is a series of disjoint intervals IUB_i^v such that $set(A_i)$ is constrained to be included in the union of the IUB_i^v. If the size of the lower bound (*i.e.*, the sum of the sizes of the ILB_i^u) becomes larger than the upper bound of $proc(A_i)$ or if the size of the upper bound (*i.e.*, the sum of the sizes of the IUB_i^v) becomes smaller than lower bound of $proc(A_i)$, a contradiction is detected. If the size of the lower bound (or of the upper bound) becomes equal to the upper bound (respectively, lower bound) of $proc(A_i)$, $set(A_i)$ receives the lower bound (respectively, the upper bound) as its final value. Minimal and maximal values of $start(A_i)$ and $end(A_i)$, *i.e.*, earliest and latest start and end times, are also maintained. Each of the following rules, considered independently one from another, is used to update the bounds of $set(A_i)$, $start(A_i)$ and $end(A_i)$. Let t be any point in time, then

$$\begin{aligned}
&t < r_i \Rightarrow t \notin set(A_i)\\
&t \in lb(set(A_i)) \Rightarrow start(A_i) \le t\\
&\bar{d}_i \le t \Rightarrow t \notin set(A_i)\\
&t \in lb(set(A_i)) \Rightarrow t < end(A_i)\\
&[\forall u < t, u \notin ub(set(A_i))] \Rightarrow t \le start(A_i)\\
&[\forall u \ge t, u \notin ub(set(A_i))] \Rightarrow end(A_i) \le t\\
&start(A_i) \le \max\{u : \exists S \subseteq ub(set(A_i)),\ |S| = p_i \wedge \min(S) = u\}\\
&end(A_i) \ge \min\{u : \exists S \subseteq ub(set(A_i)),\ |S| = p_i \wedge \max(S) = u - 1\}
\end{aligned}$$

Needless to say, whenever any of these rules leads to a situation where the lower bound of a variable is larger than its upper bound, a contradiction is detected.

In the following, we may occasionally use the notations $X(A_i, t)$ and $set(A_i)$ for an activity A_i that cannot be interrupted. In such a case, the following rules are also applied:

$$\begin{aligned}
X(A_i, t) = 0 \wedge t < eet_i \Rightarrow start(A_i) > t\\
X(A_i, t) = 0 \wedge lst_i \le t \Rightarrow end(A_i) \le t
\end{aligned}$$

7.2.2 Temporal Relations

Temporal relations between activities can be expressed by linear constraints between the start and end variables of activities. For instance, a precedence between two activities A_i and A_j is modeled by the linear constraint $end(A_i) \le start(A_j)$. Such constraints can be easily propagated using a standard arc-B-consistency algorithm [155]. In addition, a variant of Ford's algorithm (see for instance [111]) proposed by Cesta and Oddi [72] can be used to detect any inconsistency between such constraints, in time polynomial in the number of constraints (and independent of the domain sizes).

7.2.3 Resource Constraints

Resource constraints represent the fact that activities require some amount of resource throughout their execution. Given an activity A_i and a resource R whose capacity is $cap(R)$, let $cap(A_i, R)$ be the variable that represents the amount of resource R required by activity A_i. For fully elastic activities, the $cap(A_i, R)$ variable is not meaningful and we need to introduce a variable $E(A_i, R)$ that represents the *energy* required by the activity on the resource R. Note that for non-elastic activities, we have $E(A_i, R) = cap(A_i, R)proc(A_i)$. To represent a schedule, we need a variable $E(A_i, t, R)$ that denotes the number of units of the resource R used by activity A_i at time t. In all cases, we have the constraint stating that enough resource must be allocated to activities to cover the energy requirement:

$$E(A_i, R) = \sum_t E(A_i, t, R)$$

If A_i is not an elastic activity, there are some strong relations between $E(A_i, t, R)$ and $X(A_i, t)$:

$$E(A_i, t, R) = X(A_i, t)\,cap(A_i, R)$$

For elastic activities, we have a weaker relation between the variables:

$$[E(A_i, t, R) > 0] \Leftrightarrow [X(A_i, t) > 0]$$

Generally speaking, the resource constraint can be written as follows. For each point in time t

$$\sum_{i=1}^{n} E(A_i, t, R) \leq cap(R) \tag{7.2}$$

Depending on the scheduling situation, (7.2) can be rewritten. In the non-preemptive case, (7.2) leads for all times t to

$$\sum_{A_i : start(A_i) \leq t < end(A_i)} cap(A_i, R) \leq cap(R) \tag{7.3}$$

In the preemptive case, (7.2) leads for all times t to

$$\sum_{A_i : start(A_i) \leq t < end(A_i)} X(A_i, t)\,cap(A_i, R) \leq cap(R) \tag{7.4}$$

In the following, $c_{i,R}$ and $e_{i,R}$ denote the minimal capacity and energy required by the activity A_i from resource R. Finally, C_R denotes the maximal value in the domain of the resource capacity variable. Sometimes, if no confusion is possible "R" is omitted, leading to c_i, e_i, and C, respectively.

7.2.4 Extensions of the Model

Although the model presented until now covers quite a number of real-life problems, we want to include two extensions that are frequently found in industrial applications, namely *alternative resources* and *transition times and transition costs*. We refer to Chapter 15 for a discussion on other extensions found in industry.

Alternative Resources

In some scheduling situations an activity A_i can be scheduled on any one resource from a set S of resources. We say that S is the set of *alternative resources* for A_i. For each set of alternative resources a variable $altern(A_i)$ is introduced. To simplify notation, we assume that resources are numbered from 1 to m and that $altern(A_i)$ denotes the variable whose value represents the index of the resource on which A_i is executed. We remark that quite commonly the processing time of the activity depends on the resource on which the given activity is executed, *i.e.*, the resources are unrelated. Another commonly found type of constraints reasons on interdependencies of resource allocations, *e.g.*, a constraint like "if A_1 is scheduled on resource R_1 then A_3 has to be scheduled on resource R_2".

Alternative resource constraints are propagated as if A_i were split into $|domain(altern(A_i))|$ fictive activities A_i^u where each activity A_i^u requires resource R_u [153]. Following this notation r_i^u denotes the earliest start time of A_i^u, *etc.* The alternative resource constraint maintains the constructive disjunction between the alternative activities A_i^u for $u \in domain(altern(A_i))$, *i.e.*, it ensures that:

$$\begin{aligned}
r_i &= \min\{r_i^u : u \in domain(altern(A_i))\} \\
lst_i &= \max\{lst_i^u : u \in domain(altern(A_i))\} \\
eet_i &= \min\{eet_i^u : u \in domain(altern(A_i))\} \\
\bar{d}_i &= \max\{\bar{d}_i^u : u \in domain(altern(A_i))\} \\
lb(proc(A_i)) &= \min\{lb(proc(A_i^u)) : u \in domain(altern(A_i))\} \\
ub(proc(A_i)) &= \max\{ub(proc(A_i^u)) : u \in domain(altern(A_i))\}
\end{aligned}$$

Constraint propagation will deduce new bounds for alternative activities A_i^u on the alternative resource R_u. Whenever the bounds of an activity A_i^u turn out to be incoherent, the resource R_u is simply removed from the set of possible alternative resources for activity A_i, *i.e.*, $domain(altern(A_i))$ becomes $domain(altern(A_i)) - \{u\}$.

Transition Times and Transition Costs

Allahverdi, Gupta, and Aldowaisan, in reviewing the research on scheduling involving setup considerations [7], discuss the importance of scheduling with sequence dependent setups in real-life applications, and encourage researchers to work on the subject. Our experience with industrial applications also formed the basis for the motivation for considering setup times and setup costs with attention.

The setup time (transition time) between two activities A_1 and A_2 is defined as the amount of time $setup(A_1, A_2)$ that must elapse between the end of A_1 and the start of A_2, when A_1 precedes A_2. A setup cost $setupCost(A_1, A_2)$ can also be associated to the transition between A_1 and A_2. The objective of the scheduling problem can be to find a schedule that minimizes the sum of the setup costs.

Throughout the manuscript, we consider that activities subjected to setups are to be scheduled on the same machine (the semantics of setups is much more complex on resources of capacity greater than 1). However, setup considerations can be combined with alternative resources. In such a case, we have to consider that two parameters are associated to each tuple (A_i, A_j, M_u): the setup time $setup(A_i, A_j, M_u)$ and the setup cost $setupCost(A_i, A_j, M_u)$ between activities A_i and A_j if A_i and A_j are scheduled sequentially on the same machine M_u. In such a case, $start(A_j^u) \geq end(A_i^u) + setup(A_i, A_j, M_u)$. There may furthermore exist a setup time $setup(-, A_i, M_u)$ (with corresponding cost $setupCost(-, A_i, M_u)$) that has to elapse before the start of A_i when A_i is the first activity on M_u and, similarly, a teardown time $setup(A_i, -, M_u)$ (with corresponding cost $setupCost(A_i, -, M_u)$) that has to elapse after the end of A_i when A_i is the last activity on M_u.

7.2.5 Objective Function

To model the objective function, we add a variable *criterion* to the model. It is constrained to equal the value of the objective function. For most problems the objective function is a function of the end variables of the activities.

$$criterion = F(end(A_1), \ldots, end(A_n)) \tag{7.5}$$

Once all constraints of the problem are added, a common technique to look for an optimal solution is to solve successive decision variants of the problem. Several strategies can be considered to minimize the value of *criterion*. One way is to iterate on the possible values, either from the lower bound of its domain up to the upper bound until one solution is found, or from the upper bound down to the lower bound determining each time whether there still is a solution. Another way is to use a dichotomizing algorithm, where one starts by computing an initial upper bound $ub(criterion)$ and an initial lower bound $lb(criterion)$ for *criterion*. Then

1. Set $D = \left\lfloor \frac{lb(criterion) + ub(criterion)}{2} \right\rfloor$
2. Constrain *criterion* to be at most D. Then solve the resulting CSP, *i.e.*, determine a solution with $criterion \leq D$ or prove that no such solution exists. If a solution is found, set $ub(criterion)$ to the value of *criterion* in the solution; otherwise, set $lb(criterion)$ to $D + 1$.
3. Iterate steps 1 and 2 until $ub(criterion) = lb(criterion)$.

Specific propagation techniques are used to propagate resource constraints (7.2), (7.3) or (7.4). Several sections of this manuscript are dedicated to these techniques. The objective constraint (7.5) is most often a simple arithmetic expression on which arc-B-consistency can be easily achieved.

Considering the objective constraint and the resource constraints independently is not a problem when F is a "maximum" such as $C_{\max}$ or $T_{\max}$. Indeed, the upper bound on *criterion* is directly propagated on the completion time of each activity, *i.e.*, latest end times are tightened efficiently. The situation is much more complex for sum functions such as $\sum w_i C_i$, $\sum w_i T_i$, or $\sum w_i U_i$. For these functions, efficient constraint propagation techniques must take into account the resource constraints and the objective constraint simultaneously. We pay attention to this in Chapters 11 and 14.

Chapter 8

Propagation of the One-Machine Resource Constraint

In this chapter we study several methods to propagate a One-Machine resource constraint: A set of n activities $\{A_1, \ldots, A_n\}$ require the same resource of capacity 1. The propagation of resource constraints is a purely deductive process that allows to deduce inconsistencies and to tighten the temporal characteristics of activities and resources. In the non-preemptive case (Section 8.1), the earliest start times and the latest end times of activities are updated. When preemption is allowed (Section 8.2), modifications of earliest end times and latest start times also apply.

8.1 Non-Preemptive Problems

First we consider the simple Time-Table mechanism, widely used in Constraint-Based Scheduling tools, that allows to propagate the resource constraint in an incremental fashion. We then consider a disjunctive formulation of the One-Machine resource constraint that compares the temporal characteristics of pairs of activities. In Section 8.1.3, we describe the Edge-Finding propagation technique, which is extremely efficient for solving disjunctive scheduling problems like the Job-Shop. In Section 8.1.4, we present a mechanism, known as "Not-First, Not-Last", that extends the basic Edge-Finding mechanism and allows additional deductions. Finally, some extensions of these mechanisms are presented in Section 8.1.5.

8.1.1 Time-Table Constraint

A simple mechanism to propagate resource constraints in the non-preemptive case relies on an explicit data structure called "Time-Table" to maintain information about resource utilization and resource availability over time. Resource constraints are propagated in two directions: from resources to activities, to update the time bounds of activities (earliest start times and latest end times) according to the availability of resources; and from activities to resources, to update the Time-Tables according to the time bounds of activities. Although several variants exist [148, 106, 149, 202, 157] the propagation mainly consists of maintaining for any time t arc-B-consistency on the formula:

$$\sum_{i=1}^{n} E(A_i, t) \leq 1$$

Since $E(A_i, t) = X(A_i, t)$ and since $X(A_i, t)$ equals 1 if and only if $\mathit{start}(A_i) \leq t < \mathit{end}(A_i)$, this leads in turn to some adjustments of r_i, eet_i, lst_i and $\bar{d}_i$:

$$start(A_i) \geq \min\{t : ub(X(A_i, t)) = 1\}$$
$$end(A_i) \leq \max\{t : ub(X(A_i, t)) = 1\} + 1$$
$$[X(A_i, t) = 0] \wedge [t < eet_i] \Rightarrow [start(A_i) > t]$$
$$[X(A_i, t) = 0] \wedge [lst_i \leq t] \Rightarrow [end(A_i) \leq t]$$

Before Propagation	r_i	$\bar{d}_i$	p_i
A_1	0	3	2
A_2	0	4	2
Propagation 1	r_i	$\bar{d}_i$	p_i
A_1	0	3	2
A_2	2	4	2
Propagation 2	r_i	$\bar{d}_i$	p_i
A_1	0	2	2
A_2	2	4	2

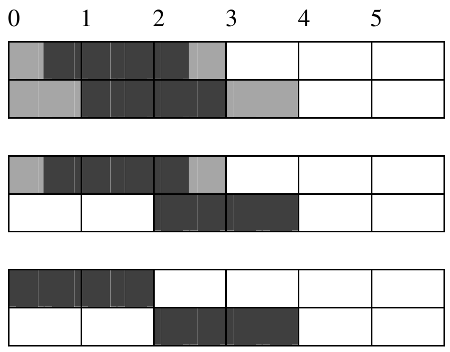


Figure 8.1: Propagation of the Time-Table constraint.

Figure 8.1 displays two activities A_1 and A_2 which require the same resource of capacity 1. The latest start time ($lst_1 = \bar{d}_1 - p_1 = 1$) of A_1 is smaller than its earliest end time ($eet_1 = r_1 + p_1 = 2$). Hence, it is guaranteed that A_1 will execute between 1 and 2. Over this period, $X(A_1, t)$ is set to 1 and the corresponding resource amount is no longer available for A_2. Since A_2 cannot be interrupted and cannot be finished before 1, the earliest start and end times of A_2 are updated to 2 and 4 (propagation 1). Then, $X(A_2, t)$ is set to 1 over the interval [2, 4), which results in a new propagation step, where the latest end time of A_1 is set to 2 (propagation 2).

8.1.2 Disjunctive Constraint Propagation

In non-preemptive scheduling, two activities A_i and A_j requiring the same machine cannot overlap in time: either A_i precedes A_j or A_j precedes A_i. If n activities require the resource, the constraint can be implemented as $n(n-1)/2$ (explicit or implicit) disjunctive constraints. As for Time-Table constraints, variants exist in the literature [96, 55, 98, 148, 203, 214, 20], but in most cases the propagation consists of maintaining arc-B-consistency on the formula:

$$[end(A_i) \leq start(A_j)] \vee [end(A_j) \leq start(A_i)]$$

Enforcing arc-B-consistency on this formula is done as follows: Whenever the earliest end time of A_i exceeds the latest start time of A_j, A_i cannot precede A_j; hence A_j must precede A_i. The time bounds of A_i and A_j are consequently updated with respect to the new temporal constraint $end(A_j) \leq start(A_i)$. Similarly, when the earliest possible end time of A_j exceeds the latest possible start time of A_i, A_j cannot precede A_i. When neither of the two activities can precede the other, a contradiction is detected. Disjunctive constraints provide more precise time bounds than the corresponding Time-Table constraints. Indeed, if an activity A_j is known to execute at some time t between the earliest start time r_i and the earliest end time eet_i of A_i, then the first disjunct of the above formula is false. Thus, A_j must precede A_i and the propagation of the disjunctive constraint implies $start(A_i) \geq eet_j > t$.

Before Propagation	r_i	$\bar{d}_i$	p_i
A_1	0	4	2
A_2	1	5	2
Propagation	r_i	$\bar{d}_i$	p_i
A_1	0	3	2
A_2	2	5	2

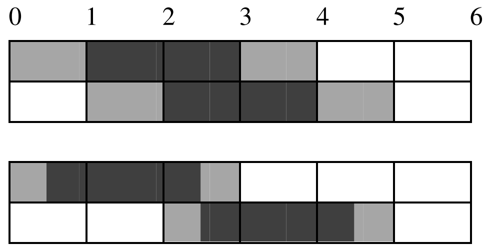


Figure 8.2: Propagation of the disjunctive constraint.

Figure 8.2 shows that disjunctive constraints may propagate more than Time-Table constraints. The earliest end time of each activity does not exceed its latest start time, so the Time-Table constraint cannot deduce anything. On the contrary, the propagation of the disjunctive constraint imposes $end(A_1) \leq start(A_2)$ which, in turn, results in updating both $\bar{d}_1$ and r_2.

8.1.3 Edge-Finding

The term "Edge-Finding" denotes both a "branching" and a "bounding" technique [8]. The branching technique consists of ordering activities that require the same resource. At each node, a set of activities Ω is selected and, for each activity $A_i \in \Omega$, a new branch is created where A_i is constrained to execute first (or last) among the activities in Ω. The bounding technique consists of deducing that some activities from a given set Ω must, can, or cannot, execute first (or last) in Ω. Such deductions lead to new ordering relations ("edges" in the graph representing the possible orderings of activities) and new time bounds, *i.e.*, strengthened earliest start times and latest end times of activities.

In the following, r_Ω and $eetmin_\Omega$ denote respectively the smallest of the earliest start times and the smallest of the earliest end times of the activities in Ω. Similarly, $\bar{d}_\Omega$ and $lstmax_\Omega$ denote respectively the largest of the latest end times and the largest of the latest start times of the activities in Ω. Finally, let p_Ω be the sum of the minimal processing times of the activities in Ω. Let $A_i \ll A_j$ ($A_i \gg A_j$) mean that A_i executes before (after) A_j and $A_i \ll \Omega$ ($A_i \gg \Omega$) mean that A_i executes before (after) all the activities in Ω. Once again, variants exist [185, 61, 62, 63, 67, 173, 51, 163, 180] but the following rules capture the "essence" of the Edge-Finding bounding technique:

$$\forall \Omega, \forall A_i \notin \Omega, [\bar{d}_{\Omega \cup \{A_i\}} - r_\Omega < p_\Omega + p_i] \Rightarrow [A_i \ll \Omega] \tag{8.1}$$

$$\forall \Omega, \forall A_i \notin \Omega, [\bar{d}_\Omega - r_{\Omega \cup \{A_i\}} < p_\Omega + p_i] \Rightarrow [A_i \gg \Omega] \tag{8.2}$$

$$\forall \Omega, \forall A_i \notin \Omega, [A_i \ll \Omega] \Rightarrow [end(A_i) \leq \min_{\emptyset \neq \Omega' \subseteq \Omega} (\bar{d}_{\Omega'} - p_{\Omega'})] \tag{8.3}$$

$$\forall \Omega, \forall A_i \notin \Omega, [A_i \gg \Omega] \Rightarrow [start(A_i) \geq \max_{\emptyset \neq \Omega' \subseteq \Omega} (r_{\Omega'} + p_{\Omega'})] \tag{8.4}$$

If n activities require the resource, there are a priori $O(n * 2^n)$ pairs (A_i, Ω) to consider. An algorithm that performs all the time-bound adjustments in $O(n^2)$ is presented in [62]. It consists of a "primal" algorithm to update earliest start times and a "dual" algorithm to update latest end times. The primal algorithm runs as follows:

- Compute Jackson's Preemptive Schedule (JPS) for the resource under consideration. JPS is the preemptive schedule obtained by applying the following priority rule: whenever the resource is free and one activity is available, schedule the activity A_i for which $\bar{d}_i$ is the smallest. If an activity A_j becomes available while A_i is in process, stop A_i and start A_j if $\bar{d}_j$ is strictly smaller than $\bar{d}_i$; otherwise continue A_i.

- For each activity A_i, compute the set Ψ of the activities which are not finished at $t = r_i$ on JPS. Let p_j^* be the residual processing time on the JPS of the activity A_j at time t. Let $\Psi_k = \{A_j \in \Psi - \{A_i\} : \bar{d}_j \leq \bar{d}_k\}$. Take the activities of Ψ in decreasing order of latest end times and select the first activity A_k such that:

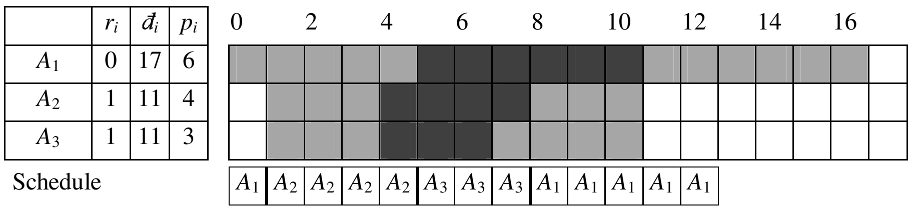
$$r_i + p_i + \sum_{A_j \in \Psi_k} p_j^* > \bar{d}_k$$

If such an activity A_k exists, then post the following constraints:

$$\begin{cases} A_i \gg \Psi_k \\ start(A_i) \geq \max_{A_j \in \Psi_k} JPS(A_j) \end{cases}$$

where $JPS(A_j)$ is the completion time of activity A_j in JPS.

Figure 8.3 presents the JPS of 3 activities. On this example, the Edge-Finding propagation algorithm deduces $start(A_1) \geq 8$ (A_1 must execute after $\{A_2, A_3\}$), when the Time-Table and the disjunctive constraint propagation algorithms deduce nothing.



	r_i	$\bar{d}_i$	p_i
A_1	0	17	6
A_2	1	11	4
A_3	1	11	3

Figure 8.3: The JPS of 3 activities.

An alternative algorithm is presented in [177, 173], also having time complexity $O(n^2)$. This algorithm is in practice easy to implement and adapt to other problems discussed later in this manuscript. The details of the algorithm are presented in Algorithm 7.

For the algorithm we assume that the set of activities $\{A_1, \ldots, A_n\}$ is ordered by ascending release dates. As such r_i is the release date of the i–th activity according to this ordering; r'_i is the adjusted value of the release date. Thanks to the introduction of the variables r'_i, one does not need to reorder activities inside the loops.

The outer iteration of the algorithm iterates over the deadlines of the activities. In the k–th iteration we consider the k-th activity in the ordering by ascending release dates and we refer to its deadline by $\bar{d}_k$. The idea then is to split the set of activities in two sets, $\Omega_{\leq}$ consisting of those activities having a deadline at most $\bar{d}_k$, and $\Omega_{>}$ consisting of those activities having a deadlines greater than $\bar{d}_k$. We will study finding updates of release dates for activities in this latter set.

After the first inner loop, *i.e.*, after line 15, the following values are calculated.

- The program variable P equals the sum of processing times in $\Omega_{\leq}$, *i.e.*, $P = p_{\Omega_{\leq}}$.
- The program variable C equals the maximal minimal end time of any subset of the set $\Omega_{\leq}$, *i.e.*, $C = \max_{\emptyset \neq \Omega' \subseteq \Omega_{\leq}} r_{\Omega'} + p_{\Omega'}$

Also the program variables C_i are calculated. They are close to the program variable C, but instead of being defined on the complete set $\Omega_{\leq}$, they are defined on the subset $\Omega_{\leq,i}$ containing the activities A_j in $\Omega_{\leq}$ that come after A_i in the ordering by ascending release dates. $\Omega_{\leq,i}$ thus is defined as $\{A_j \in \Omega_{\leq} : j \geq i\} = \{A_j \in \Omega : \bar{d}_j \leq \bar{d}_k \wedge j \geq i\}$, and C_i is defined as $\max_{\emptyset \neq \Omega' \subseteq \Omega_{\leq,i}} r_{\Omega'} + p_{\Omega'}$.

Lines 21-23 describe the deductions corresponding to rules 8.2 and 8.4 for the activity A_i at hand and all subsets of $\Omega_{\leq,i}$. Through line 19 the program variable P equals $p_{\Omega_{\leq,i}}$. So if $r_i + p_i + P$ exceeds $\bar{d}_k$, then $A_i \gg \Omega_{\leq,i}$, and thus r_i can be updated to $\max(r_i, C_i)$.

Lines 24-26 describe the deductions corresponding to rules 8.2 and 8.4 for the activity A_i at hand and all subsets of $\Omega_{\leq}$ that include at least one activity A_j that comes before A_i in the ordering by

Algorithm 7 Edge-Finding

```
for $i = 1$ to $n$ do
    $r'_i := r_i$
end for
for $k := 1$ to $n$ do
    $P := 0, C := -\infty, H := -\infty$
    for $i := n$ down to 1 do
        if $\bar{d}_i \leq \bar{d}_k$ then
            $P := P + p_i$
            $C := \max(C, r_i + P)$
            if $C > \bar{d}_k$ then
                there is no feasible schedule, exit
            end if
        end if
        $C_i := C$
    end for
    for $i := 1$ to $n$ do
        if $\bar{d}_i \leq \bar{d}_k$ then
            $H := \max(H, r_i + P)$
            $P := P - p_i$
        else
            if $r_i + P + p_i > \bar{d}_k$ then
                $r'_i := \max(r'_i, C_i)$
            end if
            if $H + p_i > \bar{d}_k$ then
                $r'_i := \max(r'_i, C)$
            end if
        end if
    end for
end for
for $i = 1$ to $n$ do
    $r_i := r'_i$
end for
```

ascending release dates, *i.e.*, for which $j < i$. Through line 18 the program variable H equals the maximal minimal end time of these subsets, *i.e.*, $H = \max_{j<i} r_j + p_{\Omega_{\leq,j}}$. Let $j_{\max} < i$ be the index for which $r_{j_{\max}} + p_{\Omega_{\leq,j_{\max}}}$ is maximal. So if $H + p_i$ exceeds $\bar{d}_k$, then $A_i \gg \Omega_{\leq,j_{\max}}$, and thus r_i can be updated to $\max(r_i, C_{j_{\max}})$. By definition of C and the C_i's, $C_{j_{\max}} = C$.

Another variant of the Edge-Finding technique is presented in [63]. It runs in $O(n \log n)$ but requires much more complex data structures. [67] presents another variant, based on the explicit definition of "task intervals." This variant runs in $O(n^3)$ in the worst case, but works in an incremental fashion. Finally, [51] and [105] propose several extensions to take setup times into account.

An interesting property of the Edge-Finding technique is established in [10] and in [163]:

Proposition 40. *Considering only the resource constraint and the current time bounds of activities, the Edge-Finding algorithm computes the smallest earliest start time at which each activity A_i could start if all the other activities were interruptible.*

As shown in [154], the Edge-Finding algorithms above may perform different deductions than the more standard disjunctive constraint propagation algorithms. Each of the two techniques performs some deductions that the other technique does not perform. In practice, an Edge-Finding algorithm is often coupled with a disjunctive constraint propagation algorithm to allow a maximal amount of constraint propagation to take place. More details on this topic can be found in Chapter 4.

8.1.4 Not-First, Not-Last

The algorithms presented in the preceding sections focus on determining whether an activity A_i must execute first (or last) in a set of activities $\Omega \cup \{A_i\}$ requiring the same resource. A natural complement consists of determining whether A_i can execute first (or last) in $\Omega \cup \{A_i\}$. If not, A_i is "Not-First" and cannot start before at least one activity in Ω is finished.

In the non-preemptive case, this leads to the following rules [185, 62, 67, 21, 154, 211, 87]:

$$\forall \Omega, \forall A_i \notin \Omega, [\bar{d}_\Omega - r_i < p_\Omega + p_i] \Rightarrow \neg(A_i \ll \Omega)$$
$$\forall \Omega, \forall A_i \notin \Omega, [\bar{d}_i - r_\Omega < p_\Omega + p_i] \Rightarrow \neg(A_i \gg \Omega)$$
$$\forall \Omega, \forall A_i \notin \Omega, \neg(A_i \ll \Omega) \Rightarrow \mathit{start}(A_i) \geq \mathit{eetmin}_\Omega$$
$$\forall \Omega, \forall A_i \notin \Omega, \neg(A_i \gg \Omega) \Rightarrow \mathit{end}(A_i) \leq \mathit{lstmax}_\Omega$$

Before Prop.	r_i	$\bar{d}_i$	p_i
A_1	1	10	2
A_2	0	5	2
A_3	2	5	1
After Prop.	r_i	$\bar{d}_i$	p_i
A_1	2	10	2
A_2	0	5	2
A_3	2	5	1

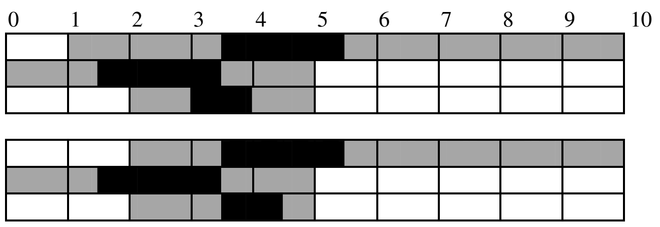


Figure 8.4: Propagation of the Not-First Not-Last rule.

In the example of Figure 8.4, the Not-First rule deduces that A_1 cannot start before 2 while the other deductive rules seen up to now deduce nothing.

The problem which consists of performing all the time-bound adjustments corresponding to the first and third rules can be called the "Not-First" problem, since it consists of updating the earliest start time of every activity A_i which cannot be first to execute in a set $\Omega \cup \{A_i\}$. Similarly, the problem which consists of performing all the time-bound adjustments corresponding to the second and fourth rules can be called the "Not-Last" problem. It consists of updating the latest end time of every activity A_i which cannot be last to execute in a set $\Omega \cup \{A_i\}$. In this section, we present the

$O(n^2)$ time and $O(n)$ space algorithm of [21] for the "Not-First" problem. The "Not-Last" problem is solved in a symmetric fashion. Alternative approaches can be found in [211, 87].

Let us first introduce some assumptions and notations. We assume that the relation $r_i + p_i \leq \bar{d}_i$ holds for every activity A_i. Otherwise, the scheduling problem clearly allows no solution and the constraint propagation process can stop. We also assume that the activities $A_1, \ldots, A_n$ which require the resource under consideration are **sorted in non-decreasing order of latest end times** (this can be done in $O(n \log n)$ time). Hence, $i \leq j$ implies $\bar{d}_i \leq \bar{d}_j$. For a given j and a given k, $\Omega(j,k)$ denotes the set of indices $m \in \{1, \ldots, k\}$ such that $r_j + p_j \leq r_m + p_m$ and $\Omega(i,j,k)$ denotes $\Omega(j,k) - \{i\}$. Hence, if i does not belong to $\Omega(j,k)$, $\Omega(i,j,k)$ is equal to $\Omega(j,k)$. Let $S_{j,k} = p_{\Omega(j,k)}$ if $j \leq k$ and $S_{j,k} = -\infty$ otherwise. Let $d_{j,k} = \min_{l \leq k}(\bar{d}_l - S_{j,l})$.

Proposition 41. *For a given j, the values $d_{j,1}, \ldots, d_{j,n}$ can be computed in $O(n)$ time.*

Proof. The values of $S_{j,k}$ and $d_{j,k}$ can be computed in constant time from the values of $S_{j,k-1}$ and $d_{j,k-1}$. One just has to test whether k verifies $r_j + p_j \leq r_k + p_k$ or not. □

Proposition 42. *If the "Not-First" rules applied to activity A_i and set Ω allow to update the earliest start time of A_i to $r_j + p_j$ then there exists an index $k \geq j$ such that the "Not-First" rules applied to activity A_i and set $\Omega(i,j,k)$ allow to update the earliest start time date of A_i to $r_j + p_j$.*

Proof. Let k be the maximal index of the activities in Ω. Ω is included in $\Omega(i,j,k)$ and $\bar{d}_\Omega$ is equal to $\bar{d}_{\Omega(i,j,k)}$. Hence the rules can be applied to A_i and $\Omega(i,j,k)$ and provide the conclusion that A_i cannot start before $r_j + p_j$ since every m in $\Omega(i,j,k)$ satisfies $r_j + p_j \leq r_m + p_m$. □

Proposition 43. *Let i and j be such that $r_i + p_i < r_j + p_j$. In this case, the "Not-First" rules allow to update the earliest start time of A_i to $r_j + p_j$ if and only if $r_i + p_i > d_{j,n}$.*

Proof. Necessary condition. Let us assume that the rules allow to update the earliest start time of A_i to $r_j + p_j$. According to Proposition 42, there exists $k \geq j$ such that $\Omega(i,j,k)$ is not empty and $\bar{d}_k - r_i < p_{\Omega(i,j,k)} + p_i$. Since $r_i + p_i < r_j + p_j$ implies that i does not belong to $\Omega(j,k)$, this implies $r_i + p_i > \bar{d}_k - S_{j,k} \geq d_{j,n}$. □

Proof. Sufficient condition. Let us assume that $r_i + p_i > d_{j,n}$. $d_{j,n}$ is finite, so there exists an index $k \geq j$ such that $d_{j,n} = \bar{d}_k - S_{j,k}$. Since i does not belong to $\Omega(j,k)$, we have $\bar{d}_k - r_i < p_{\Omega(i,j,k)} + p_i$. So, the rules allow to update the earliest start time of A_i to the value $r_j + p_j$. □

Proposition 44. *Let i and j be such that $r_i + p_i \geq r_j + p_j$. In this case, the "Not-First" rules allow to update the earliest start time of A_i to $r_j + p_j$ if and only if either $r_i + p_i > d_{j,i-1}$ or $r_i > d_{j,n}$.*

Proof. Necessary condition. Let us assume that the rules allow to update the earliest start time of A_i to $r_j + p_j$. According to Proposition 42, there exists $k \geq j$ such that $\Omega(i,j,k)$ is not empty and $\bar{d}_k - r_i < p_{\Omega(i,j,k)} + p_i$. Two cases, $k < i$ and $i < k$, can be distinguished. If $k < i$, i does not belong to $\Omega(j,k)$. This implies that $r_i + p_i > \bar{d}_k - S_{j,k} \geq d_{j,i-1}$. On the contrary, if $i < k$, i belongs to $\Omega(j,k)$. Then $p_{\Omega(j,k)} = p_{\Omega(i,j,k)} + p_i$ and $r_i > \bar{d}_k - S_{j,k} \geq d_{j,n}$. □

Proof. Sufficient condition. If $r_i + p_i > d_{j,i-1}$, $d_{j,i-1}$ is finite, so there exists an index k, not greater than i, such that $d_{j,i-1} = \bar{d}_k - S_{j,k}$. Since i does not belong to $\Omega(j,k)$, we have $\bar{d}_k - r_i < p_{\Omega(i,j,k)} + p_i$. So, the rules allow to update the earliest start time of A_i to $r_j + p_j$. Let us now assume that $r_i + p_i \leq d_{j,i-1}$ and $r_i > d_{j,n}$. Then there exists an index $k \geq j$ such that $d_{j,n} = \bar{d}_k - S_{j,k}$. Note that $k \geq i$ (otherwise, $d_{j,n} = d_{j,i-1} < r_i$ contradicts $r_i + p_i \leq d_{j,i-1}$). Consequently, i belongs to $\Omega(j,k)$. In addition, $\Omega(j,k)$ is not reduced to $\{i\}$, otherwise we would have $r_i > d_{j,n} = \bar{d}_k - S_{j,k} = \bar{d}_k - p_i \geq \bar{d}_i - p_i$ which contradicts the initial assumption that $r_i + p_i \leq \bar{d}_i$ for all i. Hence, $\Omega(i,j,k) \neq \emptyset$ satisfies $\bar{d}_k - r_i < p_{\Omega(i,j,k)} + p_i$. So, the rules allow to update the earliest start time of A_i to $r_j + p_j$. □

Algorithm 8 performs the time-bound adjustments corresponding to the "Not-First" rules.

Proposition 45. *Algorithm 8 performs the time-bound adjustments corresponding to the "Not-First" rules. It runs in $O(n^2)$ time and $O(n)$ space.*

Algorithm 8 The Not-First Algorithm

```
for $i = 1$ to $n$ do
    $r'_i := r_i$
end for
for $j = 1$ to $n$ do
    Compute $d_{j,1}, \ldots, d_{j,n}$
    for $i = 1$ to $n$ do
        if $r_i + p_i < r_j + p_j$ then
            if $r_i + p_i > d_{j,n}$ then
                $r'_i := \max(r'_i, r_j + p_j)$
            end if
        else
            if $r_i + p_i > d_{j,i-1}$ or $r_i > d_{j,n}$ then
                $r'_i := \max(r'_i, r_j + p - j)$
            end if
        end if
    end for
end for
for $i = 1$ to $n$ do
    $r_i := r'_i$
end for
```

Proof. Propositions 43 and 44 imply that the algorithm performs exactly the deductions implied by the rules. Thanks to the introduction of the variables r'_i, one does not need to reorder activities inside the loops. The algorithm runs in $O(n^2)$ steps since for each j in the outer loop, $O(n)$ steps are required to compute $d_{j,1} \ldots d_{j,n}$ and for each i in the inner loop, O(1) steps are required to perform the relevant tests. In addition, the algorithm requires a linear amount of memory space since only the values $d_{j,1} \ldots d_{j,n}$ for a given j are required. □

Let us note that when the processing times of activities are fixed, the "Not-First" and "Not-Last" rules subsume the disjunctive constraint propagation technique mentioned in Section 8.1.2. Hence, no disjunctive constraint propagation algorithm is needed when the "Not-First" algorithm above and its dual "Not-Last" algorithm are applied.

8.1.5 More Propagation

In [178] some supplementary ways of updating the earliest start times and latest end times are given. We illustrate these methods by way of an example. In Figure 8.5 we can observe that activity A_1 cannot be started at time 6. This can be seen by observing that if A_1 is started at time 6, at most one activity can be scheduled before A_1 and at most one activity can be scheduled after A_1. This means that one activity from the three activities A_2, A_3, and A_4 cannot be scheduled. In this example we can deduce that r_1 can be set to 8, being the earliest end time of any combination of two out of the three activities A_2, A_3, and A_4.

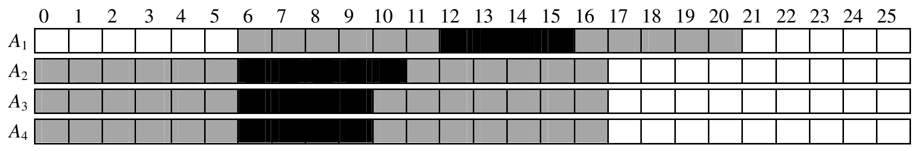


Figure 8.5: Four activities on the same machine.

More generally speaking, an activity A_i cannot start at a certain time t if there is a set $\Omega \subseteq \{A_1, \ldots, A_n\}$ of activities $A_i \notin \Omega$ and a u, $1 \leq u \leq |\Omega|$ such that at most $u-1$ activities can be scheduled before t and at most $|\Omega| - u$ activities after $t + p_i$. Let us define $eet(\Omega, u)$ as the earliest end time of any subset $\Omega' \subseteq \Omega$ such that $|\Omega'| = u$ and $lst(\Omega, u)$ as the latest start time of any subset $\Omega' \subseteq \Omega$ such that $|\Omega'| = u$. Now, if for a start time t of A_i, $t < eet(\Omega, u)$, then at most $u-1$ activities in Ω can be scheduled before t. If also $lst(\Omega, |\Omega| - u + 1) < t + p_i$, then at most $|\Omega| - u$ activities in Ω can be scheduled after $t + p_i$. In conclusion, only $|\Omega| - u + u - 1 = |\Omega| - 1$ activities in Ω can be scheduled and thus t is an inconsistent start time for A_i. As usual we seek only to update the earliest and latest start and end times of activities, which leads to the following proposition.

Proposition 46. *[178] Let A_i be an activity and let $\Omega \subseteq \{A_1, \ldots, A_n\}$ with $A_i \notin \Omega$. If there exists a u, $1 \leq u \leq |\Omega|$ such that*

$$r_i < eet(\Omega, u) \wedge lst(\Omega, |\Omega| - u + 1) < eet(A_i),$$

then r_i can be set to $eet(\Omega, u)$.

Proof. It is clear that at most $u-1$ activities in Ω can be scheduled before r_i and at most $|\Omega| - u$ activities in Ω can be scheduled after $eet(A_i)$, and thus at most $|\Omega| - 1$ activities in Ω can be scheduled in total when A_i is scheduled at r_i. Furthermore, if r_i is set to something less than $eet(\Omega, u)$ the same situation occurs. □

Now any lower bound for $eet(\Omega, u)$ and any upper bound for $lst(\Omega, |\Omega| - u + 1)$ can be used here. Using $eet(\Omega, u) = r_\Omega + \min_{\Omega' \subseteq \Omega : |\Omega'| = u} p_{\Omega'}$, and $lst(\Omega, u) = \bar{d}_\Omega - \min_{\Omega' \subseteq \Omega : |\Omega'| = u} p_{\Omega'}$, leads to an algorithm the time complexity of which is $O(n^2 \log n)$.

The following improvements on this propagation method can be implemented. Suppose that in the situation of Figure 8.5 the earliest start time of activity A_4 is updated to become 6 through propagation of some other constraint, see Figure 8.6. If we simply apply the definitions given above for $eet(\Omega, u)$ and $lst(\Omega, u)$ we do not find an update of r_1. However, we can see that A_4 cannot be scheduled before A_1 if A_1 is scheduled at r_1. With this observation we can sharpen the calculations of $eet(\Omega, 2)$ and $lst(\Omega, 2)$. As a result we can deduce that the earliest start time of A_1 can be set to 9, being the earliest end time of A_2 and A_3.

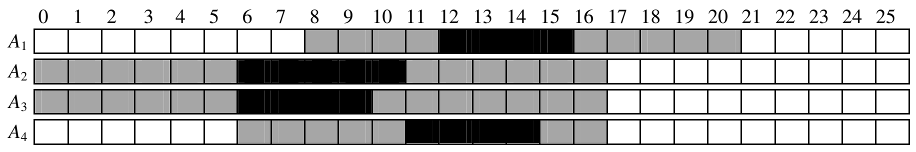


Figure 8.6: Four activities on the same machine.

Yet another way to improve the propagation can be observed in Figure 8.7 where it can be seen that with the rules given so far no update of the earliest start time of A_1 is found. However, we can observe that activity A_2 cannot be scheduled alone after A_1 when A_1 is scheduled at r_1, as the earliest end time of A_3, A_4, and A_5 is 12. Activity A_2 can, furthermore, not be scheduled after A_1 together with any other activity as the time between eet_1 and $\bar{d}_{\{A_2, A_3, A_4, A_5\}}$ is 7 and the sum of p_2 and the smallest processing time of the other activities is 8. Thus, with 12 as an upper bound on the update of r_1, we can say that A_2 is to be scheduled before A_1. As the same holds for A_3, we find that we can update r_1 to become 12.

Incorporating these improvements results in an algorithm the time complexity of which is $O(n^3)$.

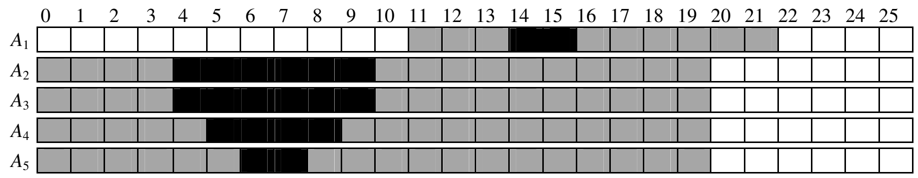


Figure 8.7: Five activities on the same machine.

8.2 Preemptive Problems

In the following sections, we study several methods to propagate the preemptive One-Machine resource constraint: A set of n interruptible activities $\{A_1, ..., A_n\}$ require the same machine. First, we show that both the Time-Table mechanism and the disjunctive constraint can be extended. We then describe a resource constraint based on network flows. In the last section, a mechanism that extends the Edge-Finding mechanism to the mixed case, *i.e.*, the case in which both interruptible and non-interruptible activities are mixed, is presented.

8.2.1 Time-Table Constraint

At the first glance, it seems that the main principle of the Time-Table mechanism directly applies to both the preemptive and the mixed case. However, an important difference appears in the relation between the variables $X(A_i, t) = E(A_i, t)$, $\mathit{set}(A_i)$, $\mathit{start}(A_i)$, $\mathit{end}(A_i)$, and $\mathit{proc}(A_i)$. The earliest start time r_i can easily be set to "the first time t at which $X(A_i, t)$ can be 1." Similarly, the latest end time $\bar{d}_i$ can easily be set to 1 + "the last time t at which $X(A_i, t)$ can be 1." However, the earliest end time eet_i is computed so that there possibly exist $\mathit{proc}(A_i)$ time points in $\mathit{set}(A_i) \cap [r_i, \mathit{eet}_i)$, and the latest start time lst_i is computed so that there possibly exist $\mathit{proc}(A_i)$ time points in $\mathit{set}(A_i) \cap [\mathit{lst}_i, \bar{d}_i)$. These additional propagation steps make the overall propagation process far more costly. Also, it is important to notice that $X(A_i, t)$ cannot be set to 1 as soon as $\mathit{lst}_i \leq t < \mathit{eet}_i$. The only situation in which $X(A_i, t)$ can be deduced to be 1 is when no more than $\mathit{proc}(A_i)$ time points can possibly belong to $\mathit{set}(A_i)$. This is unlikely to occur before decisions (choices in a search tree) are made to instantiate $\mathit{set}(A_i)$. Therefore, constraint propagation based on Time-Tables cannot prune much.

Before Prop.	r_i	eet_i	lst_i	$\bar{d}_i$	p_i
A_1 (non-int.)	0	2	1	3	2
A_2 (int.)	0	2	2	4	2
Prop. 1	r_i	eet_i	lst_i	$\bar{d}_i$	p_i
A_1 (non-int.)	0	2	1	3	2
A_2 (int.)	0	3	2	4	2

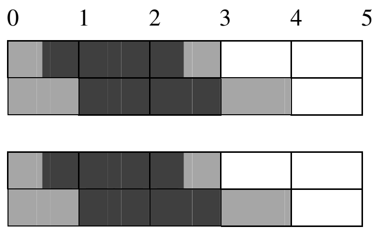


Figure 8.8: Propagation of the Time-Table constraint (mixed case). The earliest end time of A_2 is modified.

Given the data of Figure 8.1, the Time-Table mechanism cannot deduce anything if both activities can be interrupted. Figure 8.8 shows what happens when A_1 is not interruptible (non-int.) and A_2 is interruptible (int.). As in Figure 8.1, it is guaranteed that A_1 will execute between $\mathit{lst}_1 = 1$ and $\mathit{eet}_1 = 2$. Over this period, the corresponding resource amount is no longer available for A_2. The earliest end time of A_2 is then set to 3. Then the propagation process stops since there is no time point at which A_2 is guaranteed to execute.

8.2.2 Disjunctive Constraint Propagation

In the preemptive One-Machine case, the non-preemptive disjunctive constraint can be rewritten as follows:

$$\begin{array}{rl} & [\mathit{start}(A_i) + \mathit{proc}(A_i) \le \mathit{end}(A_j) - \mathit{proc}(A_j)] \\ \vee & [\mathit{start}(A_j) + \mathit{proc}(A_j) \le \mathit{end}(A_i) - \mathit{proc}(A_i)] \end{array}$$

This suggests the following preemptive disjunctive constraint:

$$\begin{array}{rl} & [\mathit{start}(A_i) + \mathit{proc}(A_i) + \mathit{proc}(A_j) \le \mathit{end}(A_i)] \\ \vee & [\mathit{start}(A_i) + \mathit{proc}(A_i) + \mathit{proc}(A_j) \le \mathit{end}(A_j)] \\ \vee & [\mathit{start}(A_j) + \mathit{proc}(A_i) + \mathit{proc}(A_j) \le \mathit{end}(A_i)] \\ \vee & [\mathit{start}(A_j) + \mathit{proc}(A_i) + \mathit{proc}(A_j) \le \mathit{end}(A_j)] \end{array}$$

which can serve as a complement to the Time-Table constraint. Arc-B-consistency is achieved on this additional constraint. Note that in the mixed case, the first (fourth) disjunct can be removed from the disjunction if A_i (respectively, A_j) cannot be interrupted.

Before Prop.	r_i	eet_i	lst_i	$đ_i$	p_i
A_1 (int.)	0	4	2	6	4
A_2 (int.)	2	3	3	4	1
Prop. 1	r_i	eet_i	lst_i	$đ_i$	p_i
A_1 (int.)	0	5	1	6	4
A_2 (int.)	2	3	3	4	1

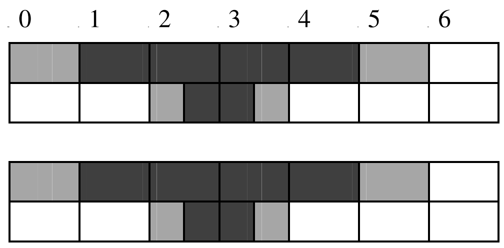


Figure 8.9: Propagation of the disjunctive constraint (preemptive case).

In the example of Figure 8.9, where both activities are interruptible, A_2 necessarily interrupts A_1. The propagation of the disjunctive constraint provides $\mathit{start}(A_1) \le 1$ and $\mathit{end}(A_1) \ge 5$.

8.2.3 Network-Flow Based Constraints

Régin [191] describes an algorithm, based on matching theory, to achieve the global consistency of the "all-different" constraint. This constraint is defined on a set of variables and constrains these variables to assume pairwise distinct values. Régin's algorithm maintains arc-consistency on the n–ary "all-different" constraint, which is shown to be more powerful than achieving arc-consistency for the $n(n-1)/2$ corresponding binary "different" constraints. Basically, Régin's algorithm consists of building a bipartite graph $G(X, Y, E)$ where X is a set of vertices corresponding to the variables of the "all-different" constraint, Y is a set of vertices corresponding to the possible values of these variables, and E is a set of edges (x, y), $x \in X$, $y \in Y$, such that $(x, y) \in E$ if and only if y is a possible value for x. As a result, the "all-different" constraint is satisfiable if and only if there exists a 0-1 function f on E such that:

$$\begin{cases} \forall x \in X, \displaystyle\sum_{y:(x,y)\in E} f(x,y) = 1 \\ \forall y \in Y, \displaystyle\sum_{x:(x,y)\in E} f(x,y) \le 1 \end{cases}$$

In addition, a given value y_j is a possible value for a given variable x_i if and only if there exists a 0-1 function $f_{i,j}$ such that:

$$\begin{cases} \forall x \in X, \displaystyle\sum_{y:(x,y)\in E} f_{i,j}(x,y) = 1 \\ \forall y \in Y, \displaystyle\sum_{x:(x,y)\in E} f_{i,j}(x,y) \le 1 \\ f_{i,j}(x_i, y_j) = 1 \end{cases}$$

The problem of finding such a function (flow) f or $f_{i,j}$ can be solved in time polynomial in the number of variables and values. In addition, the current value of f can be used to efficiently generate $f_{i,j}$, and to compute the new value of f when the domain of a variable changes. See [191, 192, 193] for details and extensions. Notice that when all activities have unitary processing times, Régin's algorithm can be directly applied. In the preemptive case, this can be generalized to activities of arbitrary processing times by seeing each activity A_i as p_i sub-activities of unitary processing times 1. Then, each sub-activity has to pick a value (the time at which the sub-activity executes) and the values of the sub-activities that require a given resource have to be pairwise distinct. However, under this naive formulation, both the number of variables and the number of values would be too high for practical use. This led us to another formulation where the nodes x in X correspond to activities, and the nodes y in Y correspond to a partition of the time horizon in q disjoint intervals $I_1 = [s_1, e_1), \ldots, I_q = [s_q, e_q)$ such that $[s_1, e_q)$ represents the complete time horizon, $e_i = s_{i+1}, (1 \leq i < q)$, and $\{s_1, \ldots, s_q, e_q\}$ includes all the time points at which the information available about $X(A_i, t)$ changes (Figure 8.10 illustrates this formulation on a small example). In particular, $\{s_1, \ldots, s_q, e_q\}$ includes all the earliest start times and latest end times of activities, but it can also include bounds of intervals over which $X(A_i, t)$ is constrained to be 1 or 0 (in this sense, the flow model is more general than preemptive Edge-Finding described in Section 8.2.4 but it does not generalize to the mixed case). E is defined as the set of pairs (x, y) such that activity x can execute during interval y. The maximal capacity $c_{\max}(x, y)$ of edge (x, y) is set to *length*(y), and the minimal capacity $c_{\min}(x, y)$ of edge (x, y) is set to *length*(y) if x is constrained to execute over y and to 0 otherwise. As a result, the preemptive resource constraint is satisfiable if and only if there exists a function f on E such that:

$$\begin{cases} \forall x \in X, \displaystyle\sum_{y:(x,y)\in E} f(x,y) = p_x \\ \forall y \in Y, \displaystyle\sum_{x:(x,y)\in E} f(x,y) \leq \mathit{length}(y) \\ \forall e \in E, c_{\min}(e) \leq f(e) \leq c_{\max}(e) \end{cases}$$

	r_i	$\bar{d}_i$	p_i
A_1	0	10	5
A_2	2	4	1
A_3	4	6	1
A_4	6	8	1

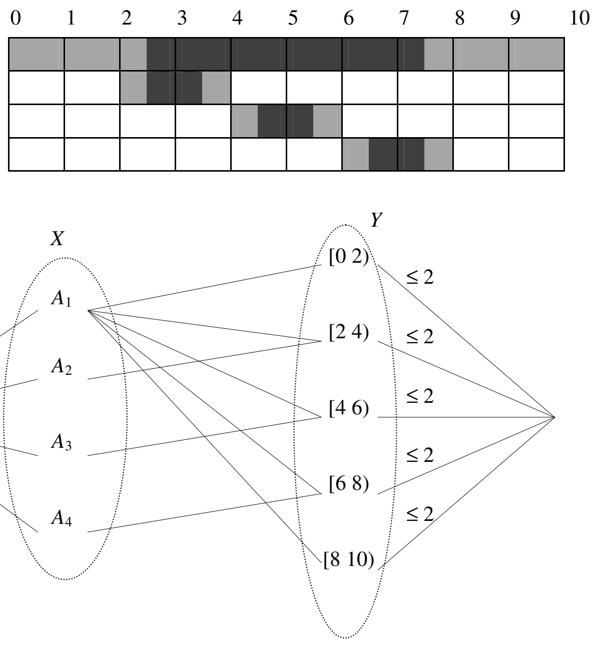


Figure 8.10: The network flow model.

Similar models are commonly used in Operations Research. For example, Federgruen and Groenevelt [100] use a more general model to solve particular polynomial scheduling problems with multiple parallel resources operating at different speeds. Following [191], what we propose below is to use network flow techniques, not only to find solutions to polynomial subproblems, but also to update the domains of the variables.

To search for compatible flow f (SCF), one can use a well-known algorithm such as Herz's algorithm, as described in [111, 5]. It runs in $O(|X| * |Y| * \sum_{x \in X} p_x)$. Alternatively, one can build a variant of Jackson's Preemptive Schedule which respects the intervals during which activities are required to execute. As shown in [10], this can be done in $O(|Y| * log(|Y|))$. This schedule is then used as an initial (possibly incompatible) flow, repaired by Herz's algorithm in $O(|X| * |Y| * F)$, where F denotes the sum, over the activities, of the sizes of the intervals included in $[r_i, \bar{d}_i)$ during which the activity A_i is not allowed to execute (for reasons that are not directly related to the use of the resource by other activities).

To reduce variable domains, the most natural generalization of Régin's algorithm consists of varying $c_{\min}(e)$ and $c_{\max}(e)$ for each edge e in turn. Algorithm 9 updates the minimal flow $c_{\min}(x,y)$ that can pass through an edge (x,y). The maximal flow $c_{\max}(x,y)$ is obtained in a similar fashion. It is proven in [10] that this adjustment of edge capacities (AEC) can be done for all edges (x,y)

Algorithm 9 Adjustment of Edge Capacities (AEC)

```
1: $u := c_{\min}(x,y)$ and $v = c_{\max}(x,y)$.
2: while $u \neq v$ do
3:    $w := \lfloor (u+v)/2 \rfloor$
4:    Search for a compatible flow with $f(x,y) \leq w$
5:    if $f$ exists then
6:       $v := f(x,y)$
7:    else
8:       $u := w+1$
9:    end if
10: end while
11: $c_{\min}(x,y) := u$
```

in $O(|X|^2|Y|H)$, where H denotes the overall time horizon $e_q - s_1$. This complexity is reached by systematically reusing the previous flow as a start point when computing the flow f with the new constraint $f(x,y) \leq w$. Then the following rules can be applied:

$$\begin{aligned}
&[c_{\max}(x,y) = 0] \Rightarrow [\forall t \in y, X(x,t) = 0] \\
&[c_{\min}(x,y) = \mathit{length}(y)] \Rightarrow [\forall t \in y, X(x,t) = 1] \\
&[c_{\min}(x,[s_i,e_i)) \neq 0] \Rightarrow [\mathit{start}(x) \leq e_i - c_{\min}(x,[s_i,e_i))] \\
&[c_{\min}(x,[s_i,e_i)) \neq 0] \Rightarrow [\mathit{end}(x) \geq s_i + c_{\min}(x,[s_i,e_i))]
\end{aligned}$$

However, SCF and AEC are not sufficient to determine the best possible time bounds for activities. Let us consider, for example, the four activities A_1, A_2, A_3, A_4 defined on Figure 8.10. In this case, $c_{\min}(A_1, I)$ remains equal to 0 for all I; yet A_1 cannot start after 3 and cannot end before 7. However, the flow model can be used to compute the best possible earliest end times. First, given x and the intervals $y_1, \ldots, y_q$ (sorted in reverse chronological order) to which x is connected, one can find the maximal integer k such that there exists a compatible flow f with $f(x,y_i) = 0$ for $1 \leq i < k$. Then, one can compute the minimal flow $f_{\min}(x,y_k)$ through (x,y_k), under the constraints $f(x,y_i) = 0$ for $1 \leq i < k$. Under these conditions, $\mathit{end}(x) \geq s_k + f_{\min}(x,[s_k,e_k))$ provides the best possible earliest end time for x. It is shown in [10] that this global update of time bounds (GUTB) can be done for all activities x in $O(|X|^2|Y|H)$. As for AEC, this complexity is reached by systematically reusing the previous flow as a start point for computing the new flow when an additional capacity constraint is added.

Let us remark that the incrementality of Herz's algorithm is a key factor for both the worst-case and the practical complexity of SCF, AEC and GUTB. Of course, strongly polynomial algorithms (with complexity independent of the schedule duration) could also be used for the search of a compatible flow [111].

8.2.4 Preemptive Edge-Finding

The Edge-Finding algorithm detailed in Section 8.1.3 can be extended to take into account the preemptive case and also the mixed case (*i.e.*, the case where interruptible and non-interruptible activities are mixed). As mentioned in Section 8.1.3, Baptiste and Martin and Shmoys [10, 163] have established an interesting property of the non-preemptive Edge-Finding technique. Considering only the resource constraint and the current time bounds of activities, the algorithm computes the earliest start time at which each activity A_i could start if all the other activities were interruptible. This suggests a logical extension of the technique to preemptive and mixed cases: for each activity A_i requiring the resource, if A_i is not interruptible, the non-preemptive Edge-Finding bound applies; if A_i is interruptible then, considering only the resource constraint and the current time bounds, it would be nice to determine the earliest start and end times between which A_i could execute if all the activities were interruptible.

Let us define $\succ$ so that $A_i \succ \Omega$ means "A_i ends after all activities in Ω" and substitute $\succ$ for $\gg$ in the rules of the primal algorithm.

$$\forall \Omega, \forall A_i \notin \Omega, [\bar{d}_\Omega - r_{\Omega \cup \{A_i\}} < p_\Omega + p_i] \Rightarrow A_i \succ \Omega$$
$$\forall \Omega, \forall A_i \notin \Omega, A_i \succ \Omega \Rightarrow [\mathit{start}(A_i) \geq \max_{\emptyset \neq \Omega' \subseteq \Omega} (r_{\Omega'} + p_{\Omega'})]$$

When A_i cannot be interrupted, these two rules remain valid (even if other activities can be interrupted) and the adjustment of r_i is the same as in the non-preemptive case. When A_i can be interrupted, the first rule is still valid but the second is not. However, the second rule can be replaced by a weaker one:

$$\forall \Omega, \forall A_i \notin \Omega, A_i \succ \Omega \Rightarrow [\mathit{end}(A_i) \geq \max_{\Omega' \subseteq \Omega} (r_{\Omega' \cup \{A_i\}} + p_{\Omega' \cup \{A_i\}})]$$

This leads to a more general primal Edge-Finding algorithm:

- Compute Jackson's Preemptive Schedule (JPS) for the resource under consideration (*cf.*, Section 8.1.3).
- For each activity A_i, compute the set Ψ of the activities which are not finished at $t = r_i$ on JPS. Let p_j^* be the residual processing time on the JPS of the activity A_j at time t. Let $\Psi_k = \{A_j \in \Psi - \{A_i\} : \bar{d}_j \leq \bar{d}_k\}$. Take the activities of Ψ in decreasing order of latest end times and select the first activity A_k such that:

  $$r_i + p_i + \sum_{A_j \in \Psi_k} p_j^* > \bar{d}_k$$

  If such an activity A_k exists, then post the following constraints:

  $$\begin{cases} A_k \succ \Psi_k & \\ \mathit{start}(A_i) \geq \max_{A_j \in \Psi_k} \mathit{JPS}(A_j) & \text{if } A_i \text{ cannot be interrupted} \\ \mathit{end}(A_i) \geq r_i + p_i + \sum_{A_j \in \Psi_k} p_j* & \text{if } A_i \text{ can be interrupted} \end{cases}$$

  where $\mathit{JPS}(A_j)$ is the completion time of activity A_j in JPS.

In the example of Figure 8.3, the algorithm above deduces $\mathit{start}(A_1) \geq 8$ if A_1 cannot be interrupted. It deduces $\mathit{end}(A_1) \geq 13$ if A_1 can be interrupted. It is proven in [10] that considering only the resource constraint and the current time bounds of activities, this algorithm computes:

- when A_i is not interruptible: the earliest time at which A_i could start if all the other activities were interruptible;

Algorithm 10 Preemptive Edge-Finding

```
for $i = 1$ to $n$ do
    $r'_i := r_i$, $eet'_i := eet_i$
end for
for $k := 1$ to $k := n$ do
    $P := 0, C := -\infty, H := -\infty$
    for $i := n$ down to $i := 1$ do
        if $\bar{d}_i \leq \bar{d}_k$ then
            $P := P + p_i$
            $C := \max(C, r_i + P)$
            if $C > \bar{d}_k$ then
                there is no feasible schedule, exit
            end if
        end if
        $C_i := C$
    end for
    for $i := 1$ to $i := n$ do
        if $\bar{d}_i \leq \bar{d}_k$ then
            $H := \max(H, r_i + P)$
            $P := P - p_i$
        else
            if $r_i + P + p_i > \bar{d}_k$ then
                if $A_i$ can be interrupted then
                    $eet'_i := \max(eet'_i, r_i + P + p_i)$
                else
                    $r'_i := \max(r'_i, C_i)$
                end if
            end if
            if $H + p_i > \bar{d}_k$ then
                if $A_i$ can be interrupted then
                    $eet'_i := \max(eet'_i, H + p_i)$
                else
                    $r'_i := \max(r'_i, C)$
                end if
            end if
        end if
    end for
end for
for $i = 1$ to $n$ do
    $r_i := r'_i$, $eet_i := eet'_i$
end for
```

- when A_i is interruptible: the earliest time at which A_i could end if all the other activities were interruptible.

Algorithm 7 can be modified in a similar fashion. In Algorithm 10, we assume that activities are sorted in increasing order of earliest start times. Earliest start times and earliest end times are adjusted inside the inner loop (lines 25, 32 and 23, 30). The proof that this algorithm is equivalent to the JPS-based algorithm follows the proof of Algorithm 7 (*cf.*, Section 8.1.3). First, if A_i cannot be interrupted, the new algorithm makes the same conclusions as Algorithm 7, so the proof in Section 8.1.3 applies to the new algorithm. Let us now assume that A_i can be interrupted. It is proven in [10] that the earliest time at which A_i could end if all the other activities could be interrupted is equal to the maximal value of $r_{\Omega\cup\{A_i\}}+p_{\Omega\cup\{A_i\}}$ for Ω triggering the Edge-Finding rules. The earliest end times computed by the new algorithm are, when they are used, equal to $r_{\Omega\cup\{A_i\}}+p_{\Omega\cup\{A_i\}}$ for such Ω. To prove that the best possible bound is reached, consider the two cases distinguished for Algorithm 7: if all activities $A_u \in \Omega$ are such that $i < u$ then either Ω or a superset of Ω is detected by the first test ($r_i + P + p_i > \bar{d}_k$); if some activity A_u of Ω is such that $u < i$ then either Ω or a superset of Ω is detected by the second test ($H + p_i > \bar{d}_k$). In both cases, a bound greater than or equal to $r_{\Omega\cup\{A_i\}}+p_{\Omega\cup\{A_i\}}$ is found.

This algorithm can be further improved:

- When A_i can be interrupted and $set(A_i)$ is known to contain a series of intervals $ILB_i^1, \ldots, ILB_i^m$, A_i can be replaced by $m+1$ activities $A_i^1, \ldots, A_i^m, A_i'$, with each A_i^l forced to execute over ILB_i^l and A_i' with the same earliest start time and latest end time as A_i and a processing time equal to
$$p_i - \sum_{l=1}^{m} length(ILB_i^l)$$
where $length(ILB_i^l)$ denotes the length of the interval ILB_i^l.

- When A_i can be interrupted and either $r_i + P = \bar{d}_k$ or $H = \bar{d}_k$ in the course of the algorithm, it is certain that A_i cannot start before $\bar{d}_k$. Hence, the algorithm can also be used to update earliest start times of interruptible activities.

Remark. When A_i has a fixed processing time and cannot be interrupted, the computation and the use of C_i and C to compute
$$\max_{\emptyset \neq \Omega' \subseteq \Omega} (r_{\Omega'} + p_{\Omega'})$$
serves only to avoid repeated iterations of the algorithm. Indeed, suppose a purely preemptive Edge-Finding algorithm is used and suppose A_i is not interruptible. The purely preemptive Edge-Finding algorithm uses the following rules:

$$\forall \Omega, \forall A_i \notin \Omega, [\bar{d}_\Omega - r_{\Omega\cup\{A_i\}} < p_\Omega + p_i] \Rightarrow [A_i \succ \Omega]$$
$$\forall \Omega, \forall A_i \notin \Omega, [A_i \succ \Omega] \Rightarrow [end(A_i) \geq \max_{\Omega' \subseteq \Omega}(r_{\Omega'\cup\{A_i\}} + p_{\Omega'\cup\{A_i\}})]$$

When constraint propagation stops, the earliest end time of A_i is set to a value eet_i such that if all activities were interruptible, there would be a schedule S of the resource such that A_i does not start before r_i and ends at eet_i. If the processing time of A_i is fixed, the propagation of the constraint $start(A_i) + proc(A_i) = end(A_i)$ guarantees that when constraint propagation stops $r_i + p_i = eet_i$. Consequently, A_i is not interrupted in S, which implies that the non-preemptive Edge-Finding algorithm cannot find a better bound for r_i.

Chapter 9

Propagation of Cumulative Constraints

Cumulative resource constraints represent the fact that activities A_i use some amount $cap(A_i)$ of resource throughout their execution. For a fully elastic activity A_i, the $cap(A_i)$ variable is not meaningful and we use a variable $E(A_i)$ that represents the global "energy" required by the activity on the resource. Of course, for a non-elastic activity, we have $E(A_i) = cap(A_i)proc(A_i)$. In all case, enough resource must be allocated to activities, *i.e.*, $E(A_i) = \sum_t E(A_i, t)$, where $E(A_i, t)$ is the amount of resource used at t by A_i. Recall that if A_i is not an elastic activity, there are some strong relations between $E(A_i, t)$ and $X(A_i, t)$: $E(A_i, t) = X(A_i, t)cap(A_i)$ For elastic activities, we have a weaker relation between the variables: $[E(A_i, t) \neq 0] \Leftrightarrow [X(A_i, t) \neq 0]$. Generally speaking, the cumulative resource constraint can be stated as follows:

$$\forall t \sum_i E(A_i, t) \leq cap$$

In the non-preemptive case, it can be rewritten as,

$$\forall t \sum_{start(A_i) \leq t < end(A_i)} cap(A_i) \leq cap$$

and in the preemptive case as,

$$\forall t \sum_{start(A_i) \leq t < end(A_i)} X(A_i, t)cap(A_i) \leq cap$$

In the following, we note c_i and e_i the minimal amount of the capacity (resp. of the energy) of the resource required by A_i. Finally we note C the maximum value in the domain of the resource capacity. In this chapter, we recall the most well-known techniques used to propagate the cumulative resource constraint. We focus on the fully elastic case in Section 9.1, on the preemptive case in Section 9.2 and, finally, on the non-preemptive case in Section 9.3.

To simplify the presentation of the theory, we sometimes refer to the "Fully Elastic Problem", to the "Preemptive Problem" and to the "Non-Preemptive Problem". They are the decision problems respectively induced by the fully elastic resource constraint, by the preemptive resource constraint and by the non-preemptive resource constraint, *i.e.*, the problems of determining whether there exists a fully elastic, preemptive and non-preemptive schedule of the given activities on a resource of given capacity.

9.1 Fully Elastic Problems

The Fully Elastic Problem is closely related to the Preemptive One-Machine Problem. Actually, Proposition 47 shows that they are identical modulo a simple transformation.

Transformation 1. *For any instance I of the Fully Elastic Problem, let $F(I)$ be the instance of the Preemptive One-Machine Problem defined by n activities $A'_1, \ldots, A'_n$ with $\forall i, r'_i = Cr_i, \bar{d}'_i = C\bar{d}_i, p'_i = c_i p_i$.*

Proposition 47. *For any instance I of the Fully Elastic Problem, there exists a feasible fully elastic schedule of I if and only if there exists a feasible preemptive schedule of $F(I)$.*

Proof. As usual, let C be the capacity of the resource R of the instance I. Let R' be the resource of the instance $F(I)$. We first prove that if there is a feasible fully elastic schedule of I, then there is a feasible preemptive schedule of $F(I)$. We build a schedule of $A'_1, \ldots, A'_n$ on R' as follows. For each time t and each activity A_i, schedule $E(A_i, t)$ units of A'_i on R' as early as possible after time Ct. It is obvious that at any time t, for any activity A_i, the number of units of A_i executed at t on R is equal to the number of units of A'_i executed between Ct and $C(t+1)$ on R' since this algorithm consists of cutting the schedule of $A_1, \ldots, A_n$ into slices of one time unit and rescheduling these slices on R'. Consequently, for any activity A'_i, exactly $c_i p_i$ units of A'_i are scheduled between Cr_i and $C\bar{d}_i$ and thus the earliest start times as well as latest end times are met. A symmetric demonstration would prove that if there is a feasible preemptive schedule of $F(I)$ then there is a feasible fully elastic schedule of I. □

Consider now Jackson's Preemptive Schedule. JPS is feasible if and only if there exists a feasible preemptive schedule. Moreover, JPS can be built in $O(n \log n)$ steps (see [55] for details). Consequently, thanks to Proposition 47 we have an $O(n \log n)$ algorithm to solve the Fully Elastic Problem. In the following, Jackson's Fully Elastic Schedule (JFES) denotes the fully elastic schedule obtained by applying JPS on the transformed instance and rescheduling slices as described in the proof of Proposition 47.

Thanks to Proposition 47 we can also adapt all the constraint propagation mechanisms described in Section 8.2 for the preemptive case to the fully elastic case. The general adjustment framework is:

1. Build the Preemptive One-Machine Problem instance $F(I)$ corresponding to the fully elastic instance I.

2. Apply a One-Machine preemptive constraint propagation algorithm (*e.g.*, an algorithm relying on Time-Table, network flow, disjunctive constraint, or Edge-Finding) on activities $A'_1, \ldots, A'_n$ of the instance $F(I)$. As explained in Section 8.2, for each activity A'_i, four time bounds can be sharpened: the earliest start time r'_i, the latest possible start time lst'_i, the earliest possible end time eet'_i, and the latest end time $\bar{d}'_i$.

3. Update the four time bounds of each A_i.

$$r_i := \left\lfloor \frac{r'_i}{C} \right\rfloor, lst_i := \left\lfloor \frac{lst'_i}{C} \right\rfloor, eet_i := \left\lceil \frac{eet'_i}{C} \right\rceil, \bar{d}_i := \left\lceil \frac{\bar{d}'_i}{C} \right\rceil$$

Note that as far as Edge-Finding is concerned, the above algorithm runs in a quadratic number of steps since steps (1) and (3) are linear and step (2) can be done in $O(n^2)$ as detailed in Section 8.2. Note that again the Edge-Finding technique provides the best possible bounds.

Proposition 48. *The time-bound adjustments made by the Edge-Finding algorithm are the best possible ones, i.e., the lower and upper bounds for the start and end time of activities can be reached by some feasible fully elastic schedules.*

Proof. The same proof applies for each of the four time bounds. We focus on the earliest end time. The basic idea is to prove that for any A_i, (*i*) there is a fully elastic schedule on which A_i can end at the earliest end time computed by the fully elastic time-bound adjustment algorithm and (*ii*) there is no fully elastic schedule on which A_i can end before the earliest end time computed by the algorithm. Both steps can be proven thanks to Transformation 1 and to the fact that the preemptive Edge-Finding algorithm computes the best possible time bounds for the Preemptive One-Machine Problem (*cf.*, Section 8.2.4). □

9.2 Preemptive Problems

Several techniques can be used to propagate cumulative preemptive resource constraints. The most simple idea is to apply the constraint propagation algorithms developed for the fully elastic case. This is of course valid since the Fully Elastic Problem can be seen as a relaxation of the Preemptive Problem. However, it might not be very efficient. The following sections are dedicated to three specific constraint propagation schemes. The first one is based on Time-Tables, the second one is an adaptation of the disjunctive constraint and finally we introduce an adjustment scheme based upon "partially elastic" schedules.

9.2.1 Time-Table Constraint

The Time-Table constraint in the preemptive cumulative case is a simple adaptation of the One-Machine preemptive case. The propagation mainly consists of maintaining arc-B-consistency on the formula:

$$\forall t \quad \sum_{start(A_i) \leq t < end(A_i)} X(A_i, t)\, cap(A_i) \leq C$$

As for the One-Machine preemptive case, the relations between the variables $X(A_i, t)$, $set(A_i)$, $start(A_i)$, $end(A_i)$, and $proc(A_i)$ have to be carefully handled (*cf.*, Section 8.2.1).

9.2.2 Disjunctive Constraint

Again the disjunctive formulation of the preemptive One-Machine resource constraint can be generalized. Consider a pair of activities (A_i, A_j) that cannot overlap in time due, for instance, to resource constraints, *i.e.*, $c_i + c_j > C$. So we have $set(A_i) \cap set(A_j) = \emptyset$ or equivalently, $\forall t, X(A_i, t) = 0 \vee X(A_j, t) = 0$.
This suggests the following preemptive disjunctive constraint:

$$\begin{array}{ll} & [cap(A_i) + cap(A_j) \leq cap] \\ \vee & [start(A_i) + proc(A_i) + proc(A_j) \leq end(A_i)] \\ \vee & [start(A_i) + proc(A_i) + proc(A_j) \leq end(A_j)] \\ \vee & [start(A_j) + proc(A_i) + proc(A_j) \leq end(A_i)] \\ \vee & [start(A_j) + proc(A_i) + proc(A_j) \leq end(A_j)] \end{array}$$

which can serve as a complement to the Time-Table constraint. Arc-B-consistency is achieved on this additional constraint.

9.2.3 Partially Elastic Relaxation

The Fully Elastic Problem is a weak relaxation of the Preemptive Problem. To try to improve the relaxation, *i.e.*, to get better bounds, we define partially elastic schedules as fully elastic schedules to which we have added two additional constraints:

$$\forall i, \forall t \in [r_i, \bar{d}_i), \sum_{x<t} E(A_i, x) \leq c_i(t - r_i)$$

$$\forall i, \forall t \in [r_i, \bar{d}_i), \sum_{t \leq x} E(A_i, x) \leq c_i(\bar{d}_i - t)$$

Consider a resource of capacity 3 and an activity with earliest start time 0, latest end time 10, processing time 8 and resource requirement 2. Both Gantt charts of Figure 9.1 correspond to feasible fully elastic schedules. The first one is not a feasible partially elastic schedule. Indeed, 9 units of the resource are used in $[0, 4)$, which is more than $2(4 - 0)$. The second one is a feasible partially elastic schedule.

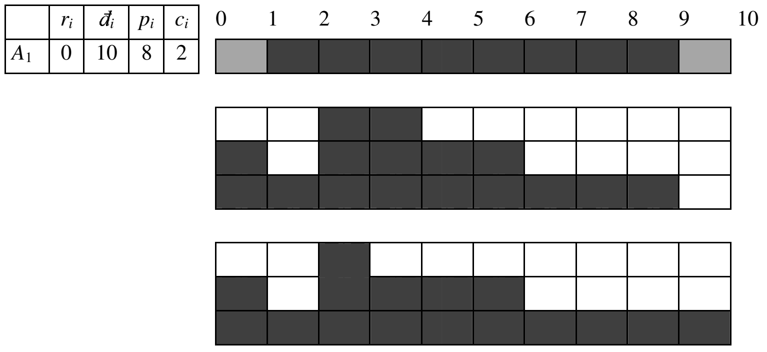


Figure 9.1: Fully and partially elastic schedules.

The main interest of the partially elastic relaxation is that it is stronger than the fully elastic relaxation and efficient constraint propagation algorithms can still be designed. First, we describe a pseudo-polynomial algorithm to test if there exists a partially elastic schedule. We then present the concept of required energy consumption, which enables us to show that the Partially Elastic Problem is equivalent to another problem for which we can provide a quadratic algorithm. In the following, "I" denotes an instance of the Partially Elastic Problem. Let us first introduce a new transformation.

Transformation 2. *Consider the instance $G(I)$ of the Fully Elastic Problem defined by replacing each activity A_i by p_i activities $A_i^1, \ldots, A_i^{p_i}$, each having a resource requirement $c_i^j = c_i$, an earliest start time $r_i^j = r_i + j - 1$, a latest end time $\bar{d}_i^j = \bar{d}_i - (p_i - j)$ and a processing time $p_i^j = 1$ (the resource capacity of $G(I)$ is C as for I).*

Jackson's Partially Elastic Schedule

Jackson's Partially Elastic Schedule (JPES) is the schedule built by scheduling each activity A_i at the time points at which the activities A_i^j are scheduled on JFES of $G(I)$. Given the definition of G, it is easy to verify that if JFES is a feasible fully elastic schedule of $G(I)$ then JPES is a feasible partially elastic schedule of I.

Proposition 49. *There exists a feasible partially elastic schedule if and only if JPES is a feasible partially elastic schedule.*

Proof. Consider a feasible partially elastic schedule S of an instance I. It is then possible to build a feasible fully elastic schedule of $G(I)$ obtained from S by a similar transformation as Transformation 2 (*i.e.*, for any activity A_i, schedule A_i^1 at the same place as the "first c_i units" of A_i on S, iterate ...). Since there is a feasible fully elastic schedule of $G(I)$, JFES is also a feasible fully elastic schedule of $G(I)$ (Proposition 47). Thus, JPES is a feasible partially elastic schedule of I. □

Since Transformation 2 is done in $O(\sum_i p_i)$ and since the Fully Elastic Problem $G(I)$ can be solved in $O(\sum_i p_i \log(\sum_i p_i))$, Proposition 49 leads to an $O(\sum_i p_i \log(\sum_i p_i))$ algorithm to test the existence of a feasible partially elastic schedule.

Energetic Reasoning

We adapt the notion of "Required Energy Consumption" defined in [158] and [159] to partially elastic activities. The required energy consumption $W_{PE}(A_i, t_1, t_2)$ of an activity over an interval $[t_1, t_2)$ is defined as follows.

$$W_{PE}(A_i, t_1, t_2) = c_i \max(0, p_i - \max(0, t_1 - r_i) - \max(0, \bar{d}_i - t_2))$$

We now define the overall required energy consumption $W_{PE}(t_1, t_2)$ over $[t_1, t_2)$ as the sum over all activities A_i of $W_{PE}(A_i, t_1, t_2)$. Note that for $t_1 = t_2, W_{PE}(t_1, t_2)$ is defined and, under the assumption $r_i + p_i \leq \bar{d}_i$, is equal to 0.

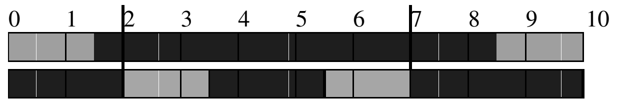


Figure 9.2: Required energy consumption for partially elastic schedules.

As shown on Figure 9.2, the required energy consumption of the activity A_1 ($r_1 = 0, \bar{d}_1 = 10, p_1 = 7, c_1 = 2$) over $[2, 7)$ is $W_{PE}(A_1, 2, 7) = 2(7 - (2 - 0) - (10 - 7)) = 4$.

Proposition 50. *There is a feasible partially elastic schedule of I if and only if for any non-empty interval $[t_1, t_2)$, $W_{PE}(t_1, t_2) \leq C(t_2 - t_1)$.*

Proof. The fact that $W_{PE}(t_1, t_2) \leq C(t_2 - t_1)$ is a necessary condition is obvious. Suppose now that there is no feasible partially elastic schedule of I. Then there is no feasible preemptive schedule of $F(G(I))$. Consequently, there is a set of activities S of $F(G(I))$ such that between the minimum earliest start time of activities in S and the maximum latest end time of activities in S there is not enough “space” to schedule all activities in S [55]:

$$\min_{A_i^j \in S} r_i^j + \sum_{A_i^j \in S} r_i^j > \max_{A_i^j \in S} \bar{d}_i^j$$

This leads to

$$\sum_{\begin{cases} r_i^j \geq Ct_1 \\ \bar{d}_i^j \leq Ct_2 \end{cases}} p_i^j > C(t_2 - t_1) \tag{9.1}$$

where Ct_1 is the minimum earliest start time in S and Ct_2 the maximum latest end time in S (recall that earliest start times and latest end times of activities in S are multiple of C). Then equation 9.1 becomes:

$$\sum_i \sum_{\begin{cases} j \in [1, p_i] \\ C(r_i + j - 1) \geq Ct_1 \\ C(\bar{d}_i - p_i + j) \leq Ct_2 \end{cases}} c_i > C(t_2 - t_1) \tag{9.2}$$

For each i, let us count the values of j in $[1, p_i]$ such that $C(r_i + j - 1) \geq Ct_1$ and $C(\bar{d}_i - p_i + j) \leq Ct_2$, *i.e.*, the number of integers in $[\max(1, t_1 + 1 - r_i), \min(p_i, t_2 + p_i - \bar{d}_i)]$. This number is equal to:

$$\begin{aligned} & \max(0, 1 + \min(p_i, t_2 + p_i - \bar{d}_i) - \max(1, t_1 + 1 - r_i)) \\ = \; & \max(0, p_i + \min(0, t_2 - \bar{d}_i) - \max(0, t_1 - r_i)) \\ = \; & \max(0, p_i - \max(0, \bar{d}_i - t_2) - \max(0, t_1 - r_i)) \end{aligned}$$

Therefore, equation 9.2 becomes $\sum_i W_{PE}(A_i, t_1, t_2) > C(t_2 - t_1)$ □

A Quadratic Algorithm

We propose a quadratic algorithm to determine whether there exists a feasible partially elastic schedule. This algorithm is derived from the algorithm used in [182] to compute the subset bound of the m–Machine Problem, *i.e.*, of $P|r_i|T_{\max}$. It consists of computing the overall required energy consumption over each interval $[r_j, \bar{d}_k)$ and to test whether this energy exceeds the energy provided by the resource over this interval. We prove that such tests guarantee the existence of a feasible partially elastic schedule. To achieve this proof, we study the slack function $S_{PE}(t_1, t_2) = C(t_2 - t_1) - W_{PE}(t_1, t_2)$.

Proposition 51. *Let t_1, t_2 be two integer values such that $t_1 < t_2$.*

- *If t_1 is not an earliest start time, then either $S_{PE}(t_1+1, t_2) < S_{PE}(t_1, t_2)$ or $S_{PE}(t_1-1, t_2) \leq S_{PE}(t_1, t_2)$.*
- *If t_2 is not a latest end time, then either $S_{PE}(t_1, t_2-1) < S_{PE}(t_1, t_2)$ or $S_{PE}(t_1, t_2+1) \leq S_{PE}(t_1, t_2)$.*

Proof. Since the two items of the proposition are symmetric, we only prove the first item. Suppose that $S_{PE}(t_1, t_2) \leq S_{PE}(t_1+1, t_2)$ and $S_{PE}(t_1, t_2) < S_{PE}(t_1-1, t_2)$. Let us then define the sets

$$\begin{aligned} \Psi &= \{i : p_i - \max(0, t_1 - r_i) - \max(0, \bar{d}_i - t_2) > 0\} \\ \Phi &= \{i : r_i \leq t_1\} \end{aligned}$$

The equation $S_{PE}(t_1, t_2) \leq S_{PE}(t_1+1, t_2)$ can be rewritten

$$-C + \sum_{i \in \Psi} c_i(-\max(0, t_1 - r_i) + \max(0, t_1 + 1 - r_i)) \geq 0$$

Since $\forall i \notin \Phi, \max(0, t_1 - r_i) = 0$ and $\max(0, t_1 + 1 - r_i) = 0$, it leads to:

$$\sum_{i \in \Psi \cap \Phi} c_i(-\max(0, t_1 - r_i) + \max(0, t_1 + 1 - r_i)) \geq C$$

which leads to

$$\sum_{i \in \Psi \cap \Phi} c_i \geq C$$

The equation $S_{PE}(t_1, t_2) < S_{PE}(t_1-1, t_2)$ can be rewritten

$$\begin{aligned} &\sum_i c_i \max(0, p_i - \max(0, t_1 - 1 - r_i) - \max(0, \bar{d}_i - t_2)) \\ -&\sum_i c_i \max(0, p_i - \max(0, t_1 - r_i) - \max(0, \bar{d}_i - t_2)) < C \end{aligned}$$

which leads to

$$\begin{aligned} &\sum_{i \in \Psi} c_i(-\max(0, t_1 - 1 - r_i) + \max(0, t_1 - r_i)) \\ +&\sum_{i \notin \Psi} c_i \max(0, p_i - \max(0, t_1 - 1 - r_i) - \max(0, \bar{d}_i - t_2)) < C \end{aligned}$$

Hence,

$$\sum_{i \in \Psi} c_i(-\max(0, t_1 - 1 - r_i) + \max(0, t_1 - r_i)) < C \tag{9.3}$$

Consider now two cases.

- If $i \in \Phi$ then $t_1 - r_i \geq 0$. Moreover, $t_1 - r_i - 1 \geq 0$ since t_1 is not an earliest start time.
- If $i \notin \Phi$ then $t_1 - r_i < 0$ and $t_1 - r_i - 1 < 0$.

(9.3) leads to $\sum_{\Psi \cap \Phi} c_i < C$, which contradicts $\sum_{\Psi \cap \Phi} c_i \geq C$. □

Proposition 52.

$$\begin{array}{ll} & [\forall r_j, \forall \bar{d}_k > r_j, S_{PE}(r_j, \bar{d}_k) \geq 0] \\ \Leftrightarrow & [\forall t_1, \forall t_2 > t_1, S_{PE}(t_1, t_2) \geq 0] \\ \Leftrightarrow & [\textit{There exists a feasible partially elastic schedule}] \end{array}$$

Proof. Note that if $t_1 < \min_i r_i$, the slack strictly increases when t_1 decreases, and if $t_2 > \max_i \bar{d}_i$, the slack strictly increases when t_2 increases. Hence, the slack function assumes a minimal value over an interval $[t_1, t_2)$ with $\min_i r_i \leq t_1 \leq t_2 \leq \max_i \bar{d}_i$. We can assume that both t_1 and t_2 are integers (if t_1 is not, the function $t \to S_{PE}(t, t_2)$ is linear between $\lfloor t_1 \rfloor$ and $\lceil t_1 \rceil$; thus either $S_{PE}(\lfloor t_1 \rfloor, t_2) \leq S_{PE}(t_1, t_2)$ or $S_{PE}(\lceil t_1 \rceil, t_2) \leq S_{PE}(t_1, t_2)$. Among the pairs of integer values (t_1, t_2) which realize the minimum of the slack, let (u_1, u_2) be the pair such that u_1 is minimal and u_2 is maximal (given u_1). We can suppose that $S_{PE}(u_1, u_2) < 0$ (otherwise the proposition holds). Consequently, $u_1 < u_2$ and thus, according to Proposition 51, either u_1 is a earliest start time or $S_{PE}(u_1 + 1, u_2) < S_{PE}(u_1, u_2)$ or $S_{PE}(u_1 - 1, u_2) \leq S_{PE}(u_1, u_2)$. Since $S_{PE}(u_1, u_2)$ is minimal, the previous inequalities lead to $S_{PE}(u_1 - 1, u_2) = S_{PE}(u_1, u_2)$; which contradicts our hypothesis on u_1. Consequently, u_1 is an earliest start time. A symmetric demonstration proves that u_2 is a latest end time. □

This proposition is of great interest since it allows us to restrict the computation of W_{PE} to intervals $[t_1, t_2)$ where t_1 is an earliest start time and t_2 is a latest end time. Before describing the algorithm, we introduce the notation $p_i^+(t_1)$ and $p_i^-(t_2)$ which denote the minimal number of time units during which A_i must execute respectively after t_1 and before t_2, *i.e.*,

$$p_i^-(t_2) = \max(0, p_i - \max(0, \bar{d}_i - t_2))$$
$$p_i^+(t_1) = \max(0, p_i - \max(0, t_1 - r_i))$$

Algorithm 12 computes $W_{PE}(t_1, t_2)$ over all relevant intervals. The basic underlying idea is that, for a given t_1, the values of t_2 at which the slope of the $t \to W_{PE}(t_1, t)$ function changes are either of the form $\bar{d}_i$ or of the form $\bar{d}_i - p_i^+(t_1)$. The procedure iterates on the relevant values of t_1 and t_2. Each time t_2 is modified, $W_{PE}(t_1, t_2)$ is computed, as well as the new slope (just after t_2) of the $t \to W_{PE}(t_1, t)$ function. Each time t_1 is modified, the set of activities with relevant $\bar{d}_i - p_i^+(t_1)$ is incrementally recomputed and resorted.

Algorithm 11 Incremental updating. Procedure *update*($Lst, t_1^{\text{old}}, t_1$)

```
1:  $move = \emptyset, no\ move = \emptyset$ (initialize two empty lists)
2:  for $A_i \in Lst$ do
3:    if $p_i^+(t_1) > 0$ then
4:      if $p_i^+(t_1) = p_i^+(t_1^{\text{old}})$ then
5:        add $A_i$ to $no\ move$
6:      else
7:        add $A_i$ to the list $move$
8:      end if
9:    end if
10: end for
11: $Lst = \text{merge}(move, no\ move)$
```

Let us detail Algorithm 12.

- Lines 1 and 2 initialize D and *Lst*. D is the array of activities sorted in increasing order of latest end times and *Lst* is the array of activities sorted in increasing order of latest start time ($\bar{d}_i - p_i$).
- The main loop (line 4) consists in an iteration over all earliest start times t_1. Notice that t_1^{old} allows to keep the previous value of t_1.

Algorithm 12 Computation of $W_{PE}(t_1, t_2)$

```
1: $D :=$ activities sorted in increasing order of $\bar{d}_i$
2: $Lst :=$ activities sorted in increasing order of $\bar{d}_i - p_i$
3: $t_1^{\text{old}} := min_i(r_i)$
4: for $t_1$ in the set of earliest start times (sorted in inc. order) do
5:    $update(Lst, t_1^{\text{old}}, t_1)$
6:    $t_1^{\text{old}} := t_1, i^D := 0, i^{Lst} := 0$
7:    $W := 0, t_2^{\text{old}} := t_1, slope := \sum_i W_{PE}(i, t_1, t_1 + 1)$
8:    while $i^{Lst} <$ length$(Lst)$ or $i^D < n$ do
9:       if $i^D < n$ then
10:          $t_2^D := \bar{d}_{D[i^D+1]}$
11:      else
12:          $t_2^D := \infty$
13:      end if
14:      if $i^{Lst} <$ length$(Lst)$ then
15:          $t_2^{Lst} := \bar{d}_{Lst[i^{Lst}+1]} - p^+_{Lst[i^{Lst}+1]}(t_1)$
16:      else
17:          $t_2^{Lst} := \infty$
18:      end if
19:      $t_2 := \min(t_2^D, t_2^{Lst})$
20:      if $t_2 = t_2^{Lst}$ then
21:          $i^{Lst} := i^{Lst} + 1, i := Lst[i^{Lst}]$
22:      else
23:          $i^D := i^D + 1, i := D[i^D]$
24:      end if
25:      if $t_1 < t_2$ then
26:          $W := W + slope(t_2 - t_2^{\text{old}})$
27:          $W_{PE}(t_1, t_2) := W$ (energy over $[t_1, t_2)$)
28:          $t_2^{\text{old}} := t_2$
29:          $\Delta := W_{PE}(i, t_1, t_2 + 1) - 2W_{PE}(i, t_1, t_2) + W_{PE}(i, t_1, t_2 - 1)$
30:          $slope := slope + \Delta$
31:      end if
32:   end while
33: end for
```

- Algorithm 11 describes the procedure *update*($Lst, t_1^{\text{old}}, t_1$) that reorders the array *Lst* in increasing order of $\bar{d}_i - p_i^+(t_1)$. This procedure will be described later on.
- Before starting the inner loop, a variable *slope* is initialized (line 7). It corresponds to the slope of the function $t \rightarrow W_{PE}(t_1, t)$ immediately after the time point t_2^{old}. t_2^{old} and *slope* are initialized line 7 and updated lines 28 and 30.
- The inner loop on t_2 (lines 8–32) consists in iterating on both arrays D and *Lst* at the same time. Because both arrays are sorted, some simple operations (lines 9–19) determine the next value of t_2. Notice that t_2 can take at most $2n$ values and that t_2 takes all the values which correspond to a latest end time. The indices i_D and i_{Lst} correspond to the current position in arrays D and *Lst* respectively.
- Lines 20 to 24 enable to increase one of the indices and to determine the activity i which has induced the current iteration.
- To understand lines 25 to 31, consider the following rewriting of $W_{PE}(A_i, t_1, t_2)$.

$$W_{PE}(A_i, t_1, t_2) = \begin{cases} 0 & \text{If } t_2 \leq \bar{d}_i - p_i^+(t_1) \\ c_i(t_2 - \bar{d}_i + p_i^+(t_1)) & \text{If } \bar{d}_i - p_i^+(t_1) < t_2 \leq \bar{d}_i \\ c_i p_i^+(t_1) & \text{If } \bar{d}_i < t_2 \end{cases}$$

  Between two consecutive values of t_2 in the inner loop, the function W_{PE} is linear. The required energy consumption between t_1 and t_2^{old} is W, as computed at the end of the previous iteration. In addition, the slope of $t \rightarrow W_{PE}(t_1, t)$ between t_2^{old} and t_2 is *slope*. So, the required energy consumption between t_1 and t_2 is $W + slope(t_2 - t_2^{\text{old}})$. Then, *slope* is updated to take into account the non-linearity of the required energy consumption of activity i at time t_2. Notice that the algorithm may execute several times lines 25–31 for the same values of t_1 and t_2 (*e.g.*, if $\bar{d}_i = \bar{d}_j$ for some i and j). In such a case, the slope is modified several times, with respect to all the activities inducing a non-linearity at time t_2.

Let us now detail the procedure *update*. This procedure reorders the array *Lst* in increasing order of $\bar{d}_i - p_i^+(t_1)$. This is done in linear time. We rely on the fact that when we move from t_1^{old} to t_1, three cases can occur.

- Either $p_i^+(t_1)$ is null and then the required energy consumption of A_i in $[t_1, t_2)$ is null; and A_i can be removed;
- Or $p_i^+(t_1) = p_i^+(t_1^{\text{old}})$ (line 4);
- Or $p_i^+(t_1) = p_i^+(t_1^{\text{old}}) - (t_1 - t_1^{\text{old}})$ (line 6).

Activities are taken in the initial order of *Lst* and are stored in either the list *no move* (second item) or in the list *move* (third item). Notice that *no move* is sorted in increasing order of $\bar{d}_i - p_i^+(t_1^{\text{old}}) = \bar{d}_i - p_i^+(t_1)$. Moreover, *move* is sorted in increasing order of $\bar{d}_i - p_i^+(t_1^{\text{old}})$ but *move* is also sorted in increasing order of $\bar{d}_i - p_i^+(t_1)$ since the difference between $p_i^+(t_1)$ and $p_i^+(t_1^{\text{old}})$ is constant for all activities in *move*. This means that we only have to merge *move* and *no move* to obtain the reordered array.

The overall algorithm runs in $O(n^2)$ since (*i*) the initial sort can be done in $O(n \log n)$, (*ii*) the procedure *update* is basically a merging procedure which runs in $O(n)$, (*iii*) the initial value of *slope* for a given t_1 is computed in $O(n)$, and (*iv*) the inner and outer loops of the algorithm both consist in $O(n)$ iterations.

Time-Bound Adjustments

In this section, we provide an adjustment scheme which relies on the required energy consumptions computed in the partially elastic case. From now on, we assume that $\forall r_j, \forall \bar{d}_k, W_{PE}(r_j, \bar{d}_k) \leq C(\bar{d}_k - r_j)$. If not, we know that there is no feasible partially elastic schedule. As for other adjustments techniques, our basic idea is to try to order activities. More precisely, given an activity A_i and an activity A_k, we examine whether A_i can end before $\bar{d}_k$.

Proposition 53. *For any activity A_j such that $r_j < \bar{d}_k$ and*

$$W_{PE}(r_j, \bar{d}_k) - W_{PE}(A_i, r_j, \bar{d}_k) + c_i p_i^+(r_j) > C(\bar{d}_k - r_j)$$

a valid lower bound of the end time of A_i is:

$$\bar{d}_k + \frac{1}{c_i}(W_{PE}(r_j, \bar{d}_k) - W_{PE}(A_i, r_j, \bar{d}_k) + c_i p_i^+(r_j) - C(\bar{d}_k - r_j))$$

Proof. Notice that $W_{PE}(r_j, \bar{d}_k) - W_{PE}(A_i, r_j, \bar{d}_k) + c_i p_i^+(r_j)$ is the overall required energy consumption over $[r_j, \bar{d}_k)$ when $\bar{d}_i$ is set to $\bar{d}_k$. If this quantity is greater than $C(\bar{d}_k - r_j)$ then A_i must end after $\bar{d}_k$. To understand the lower bound of the end time of A_i, simply notice that the numerator of the expression is the number of energy units of A_i which have to be shifted after time $\bar{d}_k$. We can divide this number of units by the amount of resource required by A_i to obtain a lower bound of the processing time required to execute these units. $\square$

As all values $W_{PE}(r_j, \bar{d}_k)$ can be computed in $O(n^2)$, this mechanism leads to a simple $O(n^3)$ algorithm. For any tuple (A_i, A_j, A_k), check whether A_i can end before $\bar{d}_k$ and in such a case compute the corresponding time-bound adjustment. This mechanism is described in Algorithm 13. The value $W = W_{PE}(r_j, \bar{d}_k)$ is computed in $O(n^3)$ at lines 4–6. This is an alternative to Algorithm 12 that computes the values W_{PE} in a quadratic amount of steps. In [11], an algorithm that performs all time-bound adjustments in $O(n^2 \log(|\{c_i\}|))$ is described, where $|\{c_i\}|$ is the number of distinct resource requirements. This complexity becomes quadratic when the algorithm is applied to an instance of the m–Machine Problem (for which $\forall i, c_i = 1$). However, this algorithm requires the use of complex data structures and is out of the scope of this chapter.

Algorithm 13 An $O(n^3)$ algorithm for partially elastic adjustments

```
1:  for $j \in \{1, \dots, n\}$ do
2:    for $k \in \{1, \dots, n\}$ with $r_j < \bar{d}_k$ do
3:      $W := 0$
4:      for $i \in \{1, \dots, n\}$ do
5:        $W := W + W_{PE}(A_i, r_j, \bar{d}_k)$
6:      end for
7:      if $W > C(\bar{d}_k - r_j)$ then
8:        there is no feasible schedule, exit
9:      else
10:       for $i \in \{1, \dots, n\}$ do
11:         $SL := C(\bar{d}_k - r_j) - W + W_{PE}(A_i, r_j, \bar{d}_k)$
12:         if $SL - c_i \max(0, p_i - \max(0, r_j - r_i)) < 0$ then
13:           $eet_i := \max(eet_i, \bar{d}_k + \lceil \frac{1}{c_i}(c_i p_i^+(r_j) - SL) \rceil)$
14:           $lst_i := \min(lst_i, r_j - \lceil \frac{1}{c_i}(c_i p_i^-(\bar{d}_k) - SL) \rceil)$
15:         end if
16:       end for
17:     end if
18:   end for
19: end for
```

The m–Machine case

In the m–Machine case, *i.e.*, when activities require exactly one unit of the resource, the partially elastic necessary condition can be compared to several results of the literature (these results are discussed in [182]). First, notice that the decision variant of the Preemptive m–Machine Problem is polynomial and can be formulated as a maximum flow problem (similar techniques as those depicted in Section 8.2 for the One-Machine Problem are used), see for instance [100] or [182]. As shown in

[182], solving this maximum flow problem leads to a worst case complexity of $O(n^3)$. The preemptive relaxation is strictly stronger than the fully and the partially elastic relaxations.

A comparison can also be made between the subset bound, a lower bound for the optimization variant of the m–Machine Problem (see for example [55, 64, 182]) and the partially elastic relaxation. An instance of the optimization variant of the m–Machine Problem consists of a set of activities characterized by an earliest start time r_i, a tail q_i and a processing time p_i, and a resource of capacity $C = m$. The objective is to find a start time assignment s_i for each activity A_i such that temporal and resource constraints are met and $\max_i(s_i + p_i + q_i)$ is minimal.

The subset bound is the maximum among all subsets J of at least C activities of the following expression, in which $R(J)$ and $Q(J)$ denote the sets of activities in J having respectively the C smallest earliest start times and the C smallest tails.

$$\frac{1}{C}\left(\sum_{A_j\in R(J)} r_j + \sum_{A_j} p_j + \sum_{A_j\in Q(J)} q_j\right)$$

[182] presents an algorithm to compute the subset bound in a quadratic number of steps. Carlier and Pinson describe an $O(n \log n + nC \log C)$ algorithm [64] which relies on a "pseudo-preemptive" relaxation of the m–Machine Problem. Notice that the subset bound can apply, thanks to a simple transformation, as a necessary condition of existence for the decision variant of the m–Machine Problem:

$$\forall J \subseteq \{A_1,\ldots,A_n\} \text{ s.t., } |J| > C, \sum_{A_j\in R(J)} r_j + \sum_{A_j\in J} p_j \le \sum_{A_j\in D(J)} \bar{d}_j$$

where $D(J)$ denotes the set of activities in J having the C largest latest end times.

Proposition 54. *In the m–Machine case, there exists a feasible partially elastic schedule if and only if the subset bound necessary condition holds.*

Proof. First, assume that there exists a feasible partially elastic schedule. Let J be any subset of at least C activities. Let $r_1, r_2, \ldots, r_C$ be the C smallest earliest start times of activities in J, and $\bar{d}_1, \bar{d}_2, \ldots, \bar{d}_C$ be the C largest latest end times of activities in J. Since before r_C, at most C activities execute, and since for each of these activities A_i at most $r_C - r_i$ units are executed before r_C, the schedule of these activities can be reorganized so that $E(A_i, t)$ is at most 1 for every $t \le r_C$. Let us now replace each activity A_i in $R(J)$ with a new activity of earliest start time r_1, latest end time $\bar{d}_i$, and processing time $p_i + (r_i - r_1)$. A feasible partially elastic schedule of the new set of activities is obtained as a simple modification of the previous schedule, by setting $E(A_i, t) = 1$ for every A_i in $R(J)$ and t in $[r_1 r_i)$. The same operation can be done between $\bar{d}_C$ and $\bar{d}_1$ for activities in $D(J)$. We have a partially elastic schedule requiring $\sum_{A_i\in J} p_i + \sum_{A_i\in R(J)}(r_i - r_1) + \sum_{A_i\in D(J)}(\bar{d}_1 - \bar{d}_i)$ units of energy between r_1 and $\bar{d}_1$. Hence, $\sum_{A_i\in J} p_i + \sum_{A_i\in R(J)}(r_i - r_1) + \sum_{A_i\in D(J)}(\bar{d}_1 - \bar{d}_i) \le C(\bar{d}_1 - r_1)$. This is equivalent to the subset bound condition for J.
We now demonstrate the other implication. Assume that the slack S_{PE} is strictly negative for some interval. Let then $[t_1, t_2)$ be the interval over which the slack is minimal and let us define J as the set of activities A_i such that $W_{PE}(A_i, t_1, t_2) > 0$. Notice that there are at least C activities in J because $r_i + p_i \le \bar{d}_i$ implies that $W_{PE}(A_i, t_1, t_2) \le t_2 - t_1$. In addition, at most C activities A_i in J are such that $r_i < t_1$. Otherwise, when t_1 is replaced by $t_1 - 1$, the slack decreases. Similarly, at most C activities A_i in J are such that $t_2 < \bar{d}_i$. Let us define $X = \{A_i \in J : r_i < t_1\}$ and $Y = \{A_i \in J : t_2 < \bar{d}_i\}$. According to the previous remark, we have $|X| \le C$ and $|Y| \le C$. Now notice that for any activity A_i in J, $W_{PE}(A_i, t_1, t_2) > 0$. Thus, we have:

$$W_{PE}(A_i, t_1, t_2) = \sum_{A_i\in J} (p_i - \max(0, t_1 - r_i) - \max(0, \bar{d}_i - t_2))$$

This can be rewritten:

$$W_{PE}(A_i, t_1, t_2) = \sum_{A_i\in J} p_i + \sum_{A_i\in X} r_i - |X|t_1 - \sum_{A_i\in Y} \bar{d}_i + |Y|t_2$$

Since $S_{PE}(t_1, t_2)$ is strictly negative, we have,

$$\sum_{A_i \in X} r_i + (C - |X|)t_1 + \sum_{A_i \in J} p_i > \sum_{A_i \in Y} \bar{d}_i + (C - |Y|)t_2$$

Moreover, because of the definition of $R(J)$ (resp. $D(J)$), and because $|X| \leq C$ and $|Y| \leq C$, we have $\sum_{A_i \in R(J)} r_j \geq \sum_{A_i \in X} r_j + (C - |X|)t_1$ and $\sum_{A_i \in D(J)} \bar{d}_j \leq \sum_{A_i \in Y} \bar{d}_j + (C - |Y|)t_2$. As a consequence, $\sum_{A_i \in R(J)} r_j + \sum_{A_i \in J} p_i > \sum_{A_i \in D(J)} \bar{d}_j$, which is exactly the subset bound necessary condition for the set J. $\square$

9.3 Non-Preemptive Problems

Many algorithms have been proposed for the propagation of the non-preemptive cumulative constraint. A representative subset of these algorithms is presented in this chapter. First, note that constraint propagation methods initially developed for the preemptive or fully elastic case can, of course, be used on non-preemptive problems. However, specific propagation methods often provide better time bounds.

9.3.1 Time-Table Constraint

In the non-preemptive cumulative case, Time-Table constraint propagation mainly consists of maintaining arc-B-consistency on the formula:

$$\forall t \sum_{start(A_i) \leq t < end(A_i)} cap(A_i) \leq C$$

Propagation between the constraints linking $X(A_i, t)$, *set*(A_i), *start*(A_i), *end*(A_i), and *proc*(A_i) is performed in the same way as in the One-Machine non-preemptive case (*cf.*, 8.1.1).

9.3.2 Disjunctive Constraint

Two activities A_i and A_j such that $c_i + c_j > C$ cannot overlap in time. Hence either A_i precedes A_j or A_j precedes A_i, *i.e.*, the disjunctive constraint holds between these activities. Arc-B-consistency is then achieved on the formula

$$\begin{aligned} & [cap(A_i) + cap(A_j) \leq cap] \\ \vee \quad & [end(A_i) \leq start(A_j)] \\ \vee \quad & [end(A_j) \leq start(A_i)] \end{aligned}$$

9.3.3 Edge-Finding

In [175, 173, 176] the Edge-Finding techniques for the One-Machine case as discussed in Section 8.1.3 are generalized to cumulative resources. The Edge-Finding techniques for the One-Machine case identify conditions for which it can be proven that an activity must be scheduled before or after a specific subset of activities on the same machine. The techniques in [175, 173, 176] extend this to cumulative resources by considering the *energy* $e_i = c_i p_i$ an activity uses. Before continuing, we recall the following notations, for each subset of activities Ω, $e_\Omega = \sum_{A_i \in \Omega} e_i$, $r_\Omega = \min_{A_i \in \Omega} r_i$, $eetmin_\Omega = \min_{A_i \in \Omega} eet_i$, $\bar{d}_\Omega = \max_{A_i \in \Omega} \bar{d}_i$ and $lstmax_\Omega = \max_{A_i \in \Omega} lst_i$.

Figure 9.3 shows four activities each requiring a capacity of 1, that must be scheduled on a resource with capacity 2. Observe that the sum of the energy of the activities is 27, whereas the available energy between times 0 and 10 is 20. This implies that A_1 cannot be started at time 0. The same holds for times 1 to 5. What can actually be deduced is that all activities in $\Omega = \{A_2, A_3, A_4\}$ end before the end of A_1.

The following proposition provides conditions such that it can be deduced that all activities of a set Ω end before the end or start after the start of an activity A_i.

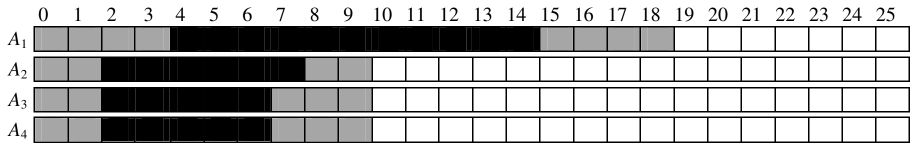


Figure 9.3: Four activities on the same resource with capacity 2.

Proposition 55. *[173] Let Ω be a set of activities and let $A_i \notin \Omega$. Then, (i) if*

$$e_{\Omega \cup \{A_i\}} > C(\bar{d}_\Omega - r_{\Omega \cup \{A_i\}}), \tag{9.4}$$

then all activities in Ω end before the end of A_i, and (ii) if

$$e_{\Omega \cup \{A_i\}} > C(\bar{d}_{\Omega \cup \{A_i\}} - r_\Omega),$$

then all activities in Ω start after the start of A_i.

Proof. (i) Suppose all activities are scheduled and an activity in Ω does not end before the end of A_i. Then the energy that is needed between $r_{\Omega \cup \{A_i\}}$ and $\bar{d}_\Omega$ is at least $e_{\Omega \cup \{A_i\}}$ which, according to (9.4), is larger than the available energy $C(\bar{d}_\Omega - r_{\Omega \cup \{A_i\}})$, which leads to a contradiction. The proof of (ii) is similar to the proof of (i)

If all activities in a set Ω end before the end of an activity A_i the earliest start time of A_i can be adjusted as follows. With $\mathcal{S}$ a schedule (and $\mathcal{S}(A_i)$ the starting time of A_i on $\mathcal{S}$) we define

$$ct(\Omega, \mathcal{S}, cp) = \min\{t \geq r_\Omega : \forall t' \geq t, \sum_{A_j \in \Omega \wedge \mathcal{S}(A_j) \leq t' < \mathcal{S}(A_j) + p_j} c_j \leq C - cp\}$$

as the earliest time after which, in schedule $\mathcal{S}$, at least a capacity of cp is available on resource μ. The minimum of $ct(\Omega, \mathcal{S}, c_i)$ over all schedules is a lower bound on r_i. The problem of calculating this lower bound is NP-hard, which can easily be seen by observing that the optimization variant of the Sequencing Problem with Release Times and Deadlines [108] is a special case of this problem by defining $c_i = C$ and $c_j = C$, for all $A_j \in \Omega$.

In [173] the following lower bound which can be computed efficiently is presented. Let Ω be a given subset of activities, then the total energy to be scheduled in $[r_\Omega, \bar{d}_\Omega)$ is e_Ω. At best an energy of $(C - c_i)(\bar{d}_\Omega - r_\Omega)$ can be scheduled, without disabling any start time of A_i. With

$$rest(\Omega, c_i) = e_\Omega - (C - c_i)(\bar{d}_\Omega - r_\Omega),$$

it is derived that if $rest(\Omega, c_i) > 0$, a lower bound on r_i equals

$$r_\Omega + \lceil rest(\Omega, c_i)/c_i \rceil.$$

The condition $rest(\Omega, c_i) > 0$ states that the total energy e_Ω that needs to be scheduled in $[r_\Omega, \bar{d}_\Omega)$ is strictly larger than the energy $(C - c_i)(\bar{d}_\Omega - r_\Omega)$, which is the energy that can be scheduled without disabling any start time of A_i.

Evidently, if all activities in a set Ω end before the end of an activity A_i then also all activities in any subset $\Omega' \subseteq \Omega$ end before the end of A_i. This leads to the observation that all start times smaller than

$$\max_{\Omega' \subseteq \Omega : \Omega' \neq \emptyset \wedge rest(\Omega', c_i) > 0} r_{\Omega'} + \lceil rest(\Omega', c_i)/c_i \rceil,$$

are inconsistent for A_i. If we do this for all possible subsets Ω, we obtain

$$\max_{\Omega \subseteq \{A_1, \ldots, A_n\} : \alpha} \max_{\Omega' \subseteq \Omega : \Omega' \neq \emptyset \wedge rest(\Omega', c_i) > 0} r_{\Omega'} + \lceil rest(\Omega', c_i)/c_i \rceil, \tag{9.5}$$

with

$$\alpha \Leftrightarrow A_i \notin \Omega \wedge C(\bar{d}_\Omega - r_{\Omega \cup \{A_i\}}) < e_{\Omega \cup \{A_i\}},$$

as a lower bound on the earliest start time of activity A_i.

Similarly, if all activities in a set Ω start after the start of an activity A_i then all end times larger than

$$\min_{\Omega' \subseteq \Omega : \Omega' \neq \emptyset \wedge rest(\Omega', c_i) > 0} \bar{d}_{\Omega'} - \lceil rest(\Omega', c_i)/c_i \rceil,$$

are inconsistent for A_i. If we do this for all subsets Ω, we obtain

$$\min_{\Omega \subseteq \{A_1, \ldots, A_n\} : \alpha} \; \min_{\Omega' \subseteq \Omega : \Omega' \neq \emptyset \wedge rest(\Omega', c_i) > 0} \bar{d}_{\Omega'} - \lceil rest(\Omega', c_i)/c_i \rceil,$$

with

$$\alpha \Leftrightarrow A_i \notin \Omega \wedge C(\bar{d}_{\Omega \cup \{A_i\}} - r_\Omega) < e_{\Omega \cup \{A_i\}}$$

as an upper bound on the latest end time of activity A_i. We remark that the lower bounds on earliest start times and upper bounds on latest end times given in by [62] for the One-Machine case (see Section 8.1.3) are special cases of the bounds given in [173].

[173] presents an algorithm for calculating these bounds with a time complexity equal to $O(n^2|\{c_i\}|)$, where $|\{c_i\}|$ is the number of distinct resource requirements. Algorithm 14 performs the same adjustments in $O(n^2)$.

As for Algorithm 7 for this algorithm we also assume that the set of activities $\{A_1, \ldots A_n\}$ is ordered by ascending earliest start times. As such r_i is the earliest start time of the i-th activity according to this ordering. Also as in other algorithms we assume that the earliest start times r_i that serve as input remain unchanged during the execution of the algorithm, *i.e.*, changes of earliest start times r_i as calculated by the algorithm are effectuated afterwards.

The outer iteration of the algorithm iterates over the latest end times of the activities. In the k-th iteration we consider the k-th activity in the ordering by ascending earliest start times and we refer to its latest end time by $\bar{d}_k$. The idea then is to split the set of activities in two sets, $\Omega_\leq$ consisting of those activities having a latest end time at most $\bar{d}_k$, and $\Omega_>$ consisting of those activities having a latest end time greater than $\bar{d}_k$. We will study finding updates of earliest start times for activities in this latter set.

Also as for the Algorithm 7 we define the set $\Omega_{\leq,i}$ as containing the activities A_j in $\Omega_\leq$ that come after A_i in the ordering by ascending earliest start times. $\Omega_{\leq,i}$ thus is defined as $\{A_j \in \Omega_\leq : j \geq i\}$ $(= \{A_j \in \Omega : \bar{d}_j \leq \bar{d}_k \wedge j \geq i\})$.

After the first inner loop, *i.e.*, after line 20, the program variable W equals the sum of energies in $\Omega_\leq$, *i.e.*, $W = e_{\Omega_\leq}$. The program variables C_i equal

$$\max_{\Omega' \subseteq \Omega_{\leq,i} : \Omega' \neq \emptyset \wedge rest(\Omega', c_i) > 0} r_{\Omega'} + \lceil rest(\Omega', c_i)/c_i \rceil.$$

To see this is true let $A_{j_{\max}} \in \Omega_{\leq,i}$ be the activity for which $e_{\Omega_{\leq,j_{\max}}} + C\, r_{j_{\max}}$ is maximal. In the first inner loop, *maxW* equals $e_{\Omega_{\leq,j_{\max}}}$ and *maxEst* equals $r_{j_{\max}}$. Through lines 13-18, one can see that if $e_{\Omega_{\leq,j_{\max}}} - (C - c_i)(\bar{d}_k - r_{j_{\max}}) > 0$ then

$$C_i = r_{j_{\max}} + \lceil (e_{\Omega_{\leq,j_{\max}}} - (C - c_i)(\bar{d}_k - r_{j_{\max}})/c_i) \rceil.$$

Now, as

$$e_{\Omega_{\leq,j_{\max}}} + C r_{j_{\max}} \geq \max_{A_{j'} \in \Omega_{\leq,i}} e_{\Omega_{\leq,j'}} + C r_{j'},$$

it is also trivially seen that

$$r_{j_{\max}} + \lceil (e_{\Omega_{\leq,j_{\max}}} - (C - c_i)(\bar{d}_k - r_{j_{\max}})/c_i) \rceil \geq$$

$$\max_{A_{j'} \in \Omega_{\leq,i}} r_{j'} + \lceil (e_{\Omega_{\leq,j'}} - (C - c_i)(\bar{d}_k - r_{j'})/c_i) \rceil.$$

>From this follows the above given definition of C_i.

Algorithm 14 Edge-Finding for cumulative resources

```
for $k := 1$ to $n$ do
  $W := 0, H := -\infty, maxW := -\infty, maxEst := \bar{d}_k$
  for $i := n$ down to $1$ do
    if $\bar{d}_i \leq \bar{d}_k$ then
      $W := W + e_i$
      if $W/C > \bar{d}_k - r_i$ then
        there is no feasible schedule, exit
      end if
      if $W + C\ r_i > maxW + C\ maxEst$ then
        $maxW := W, maxEst := r_i$
      end if
    else
      $restW := maxW - (C - c_i)(\bar{d}_k - maxEst)$
      if $restW > 0$ then
        $C_i := maxEst + \lceil restW/c_i \rceil$
      else
        $C_i := -\infty$
      end if
    end if
  end for
  for $i := 1$ to $n$ do
    if $\bar{d}_i \leq \bar{d}_k$ then
      $H := \min(H, C(\bar{d}_k - r_i) - W)$
      $W := W - e_i$
    else
      if $C(\bar{d}_k - r_i) < W + e_i$ then
        $r_i := \max(r_i, C_i)$
      end if
      if $H < e_i$ then
        $restW := maxW - (C - c_i)(\bar{d}_k - maxEst)$
        if $restW > 0$ then
          $r_i := \max(r_i, maxEst + \lceil restW/c_i \rceil)$
        end if
      end if
    end if
  end for
end for
```

Lines 26-28 describe the deductions corresponding to (9.5) for the activity A_i at hand and all subsets of $\Omega_{\leq,i}$. Through line 24 the program variable W equals $e_{\Omega_{\leq,i}}$. So if $C(\bar{d}_k - r_i)$ is smaller than $W + e_i$, then all activities in $\Omega_{\leq,i}$ end before the end of A_i, and thus r_i can be updated to $\max(r_i, C_i)$.

Lines 29-34 describe the deductions corresponding to (9.5) for the activity A_i at hand and all subsets of $\Omega_{\leq}$ that include at least one activity A_j that comes before A_i in the ordering by ascending earliest start times, *i.e.*, for which $j < i$.

Through line 23 the program variable H equals the minimal slack of these subsets, *i.e.*, $H = \min_{j<i} C(\bar{d}_k - r_j) - e_{\Omega_{\leq,j}}$. Let $j_{\min} < i$ be the index for which $C(\bar{d}_k - r_{j_{\min}}) - e_{\Omega_{\leq,j_{\min}}}$ is minimal. So if $H < e_i$ and $maxW - (C - c_i)(\bar{d}_k - maxEst) > 0$ then all activities in $\Omega_{\leq,j_{\min}}$ end before the end of A_i, and thus r_i can be updated to $\max(r_i, r_{j_{\min}} + \lceil (e_{\Omega_{\leq,j_{\min}}} - (C - c_i)(\bar{d}_k - r_{j_{\min}}))/c_i \rceil)$. By definition this means that r_i can be updated to $\max(r_i, maxEst + \lceil (maxW - (C - c_i)(\bar{d}_k - maxEst))/c_i \rceil)$.

9.3.4 Extended Edge-Finding

The following example introduces another lower bound on the earliest start time of an activity.

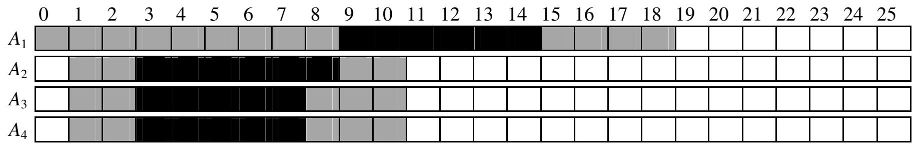


Figure 9.4: Four activities on the same resource with capacity 2.

Figure 9.4 shows four activities requiring capacity 1 which must be scheduled on a resource with capacity 2. Observe that condition (9.4) of Proposition 55 does not hold for A_1 and $\Omega = \{A_2, A_3, A_4\}$, as the sum of the energy of the activities between times 0 and 11 is 22, whereas the available energy between times 0 and 11 is also 22. Therefore, $e_{\Omega \cup \{A_1\}} = C(\bar{d}_\Omega - r_{\Omega \cup \{A_1\}})$, and no deduction can be made by using condition (9.4). However, if we focus on the interval $[r_\Omega, \bar{d}_\Omega)$, we observe that when A_1 is scheduled as early as possible, the sum of the energy of the activities between 1 and 11 is 21, whereas the available energy is 20. Again this implies that all activities in $\Omega = \{A_2, A_3, A_4\}$ end before the end of A_1.

The following proposition provides conditions such that it can be deduced that all activities of a set Ω end before the end or start after the start of an activity A_i.

Proposition 56. *[173] Let Ω be a set of activities and let $A_i \notin \Omega$. Then (i) if*

$$r_i \leq r_\Omega < eet_i, \text{ and}$$

$$e_\Omega + c_i(eet_i - r_\Omega) > C(\bar{d}_\Omega - r_\Omega), \tag{9.6}$$

then all activities in Ω end before the end of A_i, and (ii) if

$$lst_i < \bar{d}_\Omega \leq \bar{d}_i, \text{ and}$$

$$e_\Omega + c_i(\bar{d}_\Omega - lst_i) > C(\bar{d}_\Omega - r_\Omega),$$

then all activities in Ω start after the start of A_i.

Proof. (i) Suppose all activities are scheduled and an activity in Ω does not end before the end of A_i. Then the energy that is occupied between r_Ω and $\bar{d}_\Omega$ is at least $e_\Omega + c_i(eet_i - r_\Omega)$ which, according to (9.6), is larger than the available energy $C(\bar{d}_\Omega - r_\Omega)$, which leads to a contradiction. The proof of (ii) is similar to the proof of (i)

If all activities in a set Ω end before the end of an activity A_i then all start times smaller than

$$\max_{\Omega' \subseteq \Omega : \Omega' \neq \emptyset \wedge rest(\Omega', c_i) > 0} r_{\Omega'} + \lceil rest(\Omega', c_i)/c_i \rceil,$$

are inconsistent for A_i. If we do this for all subsets Ω, we obtain

$$\max_{\Omega \subseteq \{A_1,\ldots,A_n\} : \alpha} \; \max_{\Omega' \subseteq \Omega : \Omega' \neq \emptyset \wedge rest(\Omega', c_i) > 0} r_{\Omega'} + \lceil rest(\Omega', c_i)/c_i \rceil,$$

with

$$\begin{aligned} \alpha \quad \Leftrightarrow \quad & A_i \notin \Omega \wedge r_i \leq r_\Omega < eet_i \wedge \\ & e_\Omega + c_i(eet_i - r_\Omega) > C(\bar{d}_\Omega - r_\Omega), \end{aligned}$$

as a lower bound on the earliest start time of activity A_i.

Similarly, if all activities in a set Ω start after the start of an activity A_i, then all end times larger than

$$\min_{\Omega' \subseteq \Omega : \Omega' \neq \emptyset \wedge rest(\Omega', c_i) > 0} \bar{d}_{\Omega'} - \lceil rest(\Omega', c_i)/c_i \rceil,$$

are inconsistent for A_i. If we do this for all subsets Ω, we obtain

$$\min_{\Omega \subseteq \{A_1,\ldots,A_n\} : \alpha} \; \min_{\Omega' \subseteq \Omega : \Omega' \neq \emptyset \wedge rest(\Omega', c_i) > 0} \bar{d}_{\Omega'} - \lceil rest(\Omega', c_i)/c_i \rceil,$$

with

$$\begin{aligned} \alpha \quad \Leftrightarrow \quad & A_i \notin \Omega \wedge lst_i < \bar{d}_\Omega \leq \bar{d}_i \wedge \\ & e_\Omega + c_i(\bar{d}_\Omega - lst_i) > C(\bar{d}_\Omega - r_\Omega), \end{aligned}$$

as an upper bound on the latest end time of activity A_i.

[174] presents an algorithm for calculating these bounds with a time complexity equal to $O(n^3)$.

9.3.5 Not-First, Not-Last

Also in [175, 173, 176] the following proposition is given that provides conditions for which an activity A_i must be scheduled either after at least one activity or before at least one activity of a set Ω of activities that are all to be scheduled on the same cumulative resource.

Proposition 57. *[173] Let Ω be a set of activities and let $A_i \notin \Omega$. Then, (i) if*

$$r_\Omega \leq r_i < eetmin_\Omega, \text{ and}$$

$$e_\Omega + c_i(\min(eet_i, \bar{d}_\Omega) - r_\Omega) > C(\bar{d}_\Omega - r_\Omega), \tag{9.7}$$

then all start times smaller than $eetmin_\Omega$, are inconsistent, and (ii) if

$$lstmax_\Omega < \bar{d}_i \leq \bar{d}_\Omega, \text{ and}$$

$$e_\Omega + c_i(\bar{d}_\Omega - \max(lst_i, r_\Omega)) > C(\bar{d}_\Omega - r_\Omega),$$

then all end times larger than $lstmax_\Omega$, are inconsistent.

Proof. (i) If A_i is started at time r_i, then no activity of Ω can be scheduled to be completed before A_i as $r_i < eetmin_\Omega$. Thus an energy $c_i(r_i - r_\Omega)$ is occupied by no activity. Then the energy the activities in $\Omega \cup \{A_i\}$ occupy in $[r_\Omega, \bar{d}_\Omega)$ is

$$e_\Omega + c_i(\min(eet_i, \bar{d}_\Omega) - r_\Omega),$$

which, with (9.7) is larger than the available energy $C(\bar{d}_\Omega - r_\Omega)$. Thus, r_i is inconsistent for A_i, and the same holds for all start times smaller than $eetmin_\Omega$. The proof of (ii) is similar to the proof of (i)

From Proposition 57 it is derived that

$$\max_{\Omega \subseteq \{A_1,\ldots,A_n\}:\alpha} \quad eetmin_\Omega,$$

with

$$\begin{aligned} \alpha \;\Leftrightarrow\; & A_i \notin \Omega \wedge r_\Omega \le r_i < eetmin_\Omega \wedge \\ & e_\Omega + c_i(\min(eet_i, \bar{d}_\Omega) - r_\Omega) > C(\bar{d}_\Omega - r_\Omega), \end{aligned}$$

is a lower bound on the earliest start time of activity A_i. Similarly, it is derived that

$$\min_{\Omega \subseteq \{A_1,\ldots,A_n\}:\alpha} \quad lstmax_\Omega,$$

with

$$\begin{aligned} \alpha \;\Leftrightarrow\; & A_i \notin \Omega \wedge lstmax_\Omega < \bar{d}_i \le \bar{d}_\Omega \wedge \\ & e_\Omega + c_i(\bar{d}_\Omega - \max(lst_i, r_\Omega)) > C(\bar{d}_\Omega - r_\Omega), \end{aligned}$$

is an upper bound on the latest end time of activity A_i.

[174] presents an algorithm for calculating these bounds with a time complexity equal to $O(n^3)$.

9.3.6 Energetic Reasoning

Necessary Condition of Existence

The required energy consumption as defined in Section 9.2.3 is valid whether activities can be interrupted or not. In fact, [97] and [159] propose a sharper definition of the required energy consumption that takes into account the fact that activities cannot be interrupted. This leads to a new necessary condition, called the “Left-Shift / Right-Shift” necessary condition, based on energetic reasoning as defined in [97, 159]. Given an activity A_i and a time interval $[t_1, t_2), W_{Sh}(A_i, t_1, t_2)$, the “Left-Shift / Right-Shift” required energy consumption of A_i over $[t_1, t_2)$ is c_i times the minimum of the three following durations.

- $t_2 - t_1$, the length of the interval;
- $p_i^+(t_1) = \max(0, p_i - \max(0, t_1 - r_i))$, the number of time units during which A_i executes after time t_1 if A_i is left-shifted, *i.e.*, scheduled as soon as possible;
- $p_i^-(t_2) = \max(0, p_i - \max(0, \bar{d}_i - t_2))$, the number of time units during which A_i executes before time t_2 if A_i is right-shifted, *i.e.*, scheduled as late as possible.

This leads to $W_{Sh}(A_i, t_1, t_2) = c_i \min(t_2 - t_1, p_i^+(t_1), p_i^-(t_2))$. The required energy consumption of A_1 over $[2, 7)$ on Figure 9.5 is 8. Indeed, at least 4 time units of A_1 have to be executed in $[2, 7)$; *i.e.*, $W_{Sh}(A, 2, 7) = 2\min(5, 5, 4) = 8$.

We can now define the Left-Shift / Right-Shift overall required energy consumption $W_{Sh}(t_1, t_2)$ over an interval $[t_1, t_2)$ as the sum over all activities A_i of $W_{Sh}(A_i, t_1, t_2)$. We can also define the Left-Shift / Right-Shift slack over $[t_1, t_2)$: $S_{Sh}(t_1, t_2) = C(t_2 - t_1) - W_{Sh}(t_1, t_2)$. It is obvious that if there is a feasible schedule then $\forall t_1, \forall t_2 \ge t_1, S_{Sh}(t_1, t_2) \ge 0$.

In Section 9.2.3, we showed that for the partially elastic relaxation, it was sufficient to calculate the slack only for those intervals $[t_1, t_2)$ that are in the Cartesian product of the set of earliest start times and of the set of latest end times. In this section we show that in the Left-Shift / Right-Shift case, a larger number of intervals must be considered. On top of that, we provide a precise

	r_i	$\bar{d}_i$	p_i	c_i
A_1	0	10	7	2

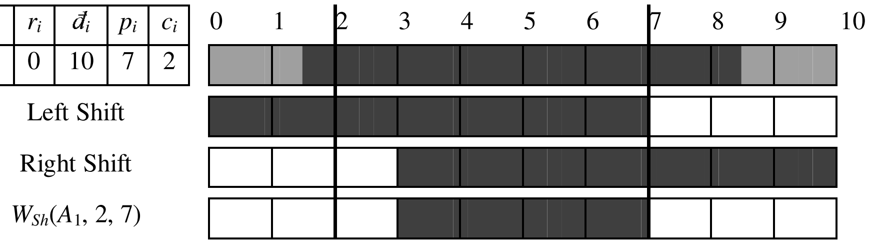


Figure 9.5: Left-Shift / Right-Shift.

characterization of the set of intervals for which the slack needs to be calculated to guarantee that no interval with negative slack exists.

Let us define the sets O_1, O_2 and $O(t)$.

$$\begin{aligned} O_1 &= \{r_i, 1 \leq i \leq n\} \cup \{\bar{d}_i - p_i, 1 \leq i \leq n\} \cup \{r_i + p_i, 1 \leq i \leq n\} \\ O_2 &= \{\bar{d}_i, 1 \leq i \leq n\} \cup \{r_i + p_i, 1 \leq i \leq n\} \cup \{\bar{d}_i - p_i, 1 \leq i \leq n\} \\ O(t) &= \{r_i + \bar{d}_i - t, 1 \leq i \leq n\} \end{aligned}$$

Proposition 58.

$$\begin{aligned} &[\forall t_1, \forall t_2 \geq t_1, S_{Sh}(t_1, t_2) \geq 0] \\ \Leftrightarrow &\begin{cases} \forall s \in O_1, \forall e \in O_2, e \geq s, S_{Sh}(s, e) \geq 0 \\ \forall s \in O_1, \forall e \in O(s), e \geq s, S_{Sh}(s, e) \geq 0 \\ \forall e \in O_2, \forall s \in O(e), e \geq s, S_{Sh}(s, e) \geq 0 \end{cases} \end{aligned}$$

To prove Proposition 58, we first need to prove some technical properties of W_{Sh} (Propositions 59, 60, 61). In the following, we consider that W_{Sh} is defined on $\mathbb{R}$ and equals 0 when $t_2 \leq t_1$. We also assume $r_i + p_i \leq \bar{d}_i$ for all i (otherwise a contradiction is obviously detected by constraint propagation).

Proposition 59. *Let A_i be an activity and $(t_1, t_2) \in \mathbb{R}^2$ with $t_1 < t_2$. If $t_1 \notin \{r_i, \bar{d}_i - p_i, r_i + p_i\}$ and if $t_2 \notin \{\bar{d}_i, \bar{d}_i - p_i, r_i + p_i\}$, then $\Phi(h) = W_{Sh}(A_i, t_1 + h, t_2 - h)$ is linear around 0.*

Proof. $\Phi(h)$ can be rewritten

$$\Phi(h) = c_i \max(0, \min(t_2 - t_1 - 2h, p_i, r_i + p_i - t_1 - h, t_2 - \bar{d}_i + p_i - h)).$$

Each of the terms 0, $t_2 - t_1 - 2h$, $p_i, r_i + p_i - t_1 - h$, $t_2 - \bar{d}_i + p_i - h$ is linear in h and if for $h = 0$, one term only realizes $\Phi(0)$, we can be sure that a small perturbation of h will have a linear effect. Assume two terms are equal and realize $\Phi(0)$. Since there are five terms, this leads us to distinguish ten cases.

1. $\Phi(0)/c_i = 0 = t_2 - t_1$,
2. $\Phi(0)/c_i = 0 = p_i$,
3. $\Phi(0)/c_i = 0 = r_i + p_i - t_1$,
4. $\Phi(0)/c_i = 0 = t_2 - \bar{d}_i + p_i$,
5. $\Phi(0)/c_i = t_2 - t_1 = p_i$,
6. $\Phi(0)/c_i = t_2 - t_1 = r_i + p_i - t_1$,
7. $\Phi(0)/c_i = t_2 - t_1 = t_2 - \bar{d}_i + p_i$,
8. $\Phi(0)/c_i = p_i = r_i + p_i - t_1$,

9. $\Phi(0)/c_i = p_i = t_2 - \bar{d}_i + p_i$,
10. $\Phi(0)/c_i = r_i + p_i - t_1 = t_2 - \bar{d}_i + p_i$.

According to our hypotheses, all cases are impossible except (5) and (10).

- We claim that case (5) cannot occur. Since $t_2 - t_1 = p_i$ and since this value is equal to $\Phi(0)/c_i$, we have $p_i < r_i + p_i - t_1$ and $p_i < t_2 - \bar{d}_i + p_i$ (equality cannot occur because of our hypotheses). Thus, $t_2 - t_1 = p_i > \bar{d}_i - r_i$; which contradicts $r_i + p_i \leq \bar{d}_i$.
- If (10) holds, then $r_i + p_i - t_1 - h = t_2 - \bar{d}_i + p_i - h$. We can moreover suppose that these two terms are the only ones to realize $\Phi(0)$ (otherwise one of the previous cases would occur). Around 0, $\Phi(h)$ can be rewritten $c_i(r_i + p_i - t_1 - h)$; which is linear.

□

Proposition 60. *Let A_i be an activity and $(t_1, t_2) \in \mathbb{R}^2$ with $t_1 < t_2$ and $t_2 \notin \{\bar{d}_i, \bar{d}_i - p_i, r_i + p_i, r_i + \bar{d}_i - t_1\}$, then $\Theta(h) = W_{Sh}(A_i, t_1, t_2 - h)$ is linear around* 0.

Proof. $\Theta(h)$ can be rewritten

$$\Theta(h) = c_i \max(0, \min(t_2 - t_1 - h, p_i, r_i + p_i - t_1, t_2 - \bar{d}_i + p_i - h))$$

Each of the terms is linear in h and if for $h = 0$, one term only realizes $\Theta(h)$, we can be sure that a small perturbation of h will have a linear effect. If two terms are equal and realize $\Theta(0)$ then either

1. $\Theta(0)/c_i = 0 = t_2 - t_1$ or
2. $\Theta(0)/c_i = 0 = p_i$ or
3. $\Theta(0)/c_i = 0 = r_i + p_i - t_1$ or
4. $\Theta(0)/c_i = 0 = t_2 - \bar{d}_i + p_i$ or
5. $\Theta(0)/c_i = t_2 - t_1 = p_i$ or
6. $\Theta(0)/c_i = t_2 - t_1 = r_i + p_i - t_1$ or
7. $\Theta(0)/c_i = t_2 - t_1 = t_2 - \bar{d}_i + p_i$ or
8. $\Theta(0)/c_i = p_i = r_i + p_i - t_1$ or
9. $\Theta(0)/c_i = p_i = t_2 - \bar{d}_i + p_i$ or
10. $\Theta(0)/c_i = r_i + p_i - t_1 = t_2 - \bar{d}_i + p_i$.

According to our hypotheses, all cases are impossible except (3), (5), (7), (8).

- If $t_1 = r_i + p_i$ (3) then $\forall h, \Theta(h) = 0$.
- Case (5) cannot occur otherwise we would have $p_i \leq r_i + p_i - t_1$ and $p_i < t_2 - \bar{d}_i + p_i$; which contradicts $r_i + p_i \leq \bar{d}_i$.
- If $t_2 - t_1 = t_2 - \bar{d}_i + p_i = \Theta(0)/c_i$ (7) then $\forall h, \Theta(h) = c_i \max(0, \min(t_2 - t_1 - h, p_i, r_i + p_i - t_1))$. We can moreover suppose that $t_2 - t_1$ is the only term in the new expression to realize $\Theta(0)$ (otherwise case (1), (5) or (6) would occur) and thus $\Theta(h) = c_i(t_2 - t_1 - h)$ around 0.
- If $p_i = r_i + p_i - t_1 = \Theta(0)/c_i$ (8) then $\Theta(h) = c_i \max(0, \min(t_2 - r_i - h, p_i, t_2 - \bar{d}_i + p_i - h))$. Since $\Theta(0) = c_i p_i > 0$, around 0 we must have $\Theta(h) = c_i \min(t_2 - r_i - h, p_i, t_2 - \bar{d}_i + p_i - h)$. Moreover, since $r_i \leq \bar{d}_i - p_i$, around 0 we have $\Theta(h) = c_i \min(p_i, t_2 - \bar{d}_i + p_i - h)$. Finally, we know that $t_2 \neq \bar{d}_i$ thus $p_i < t_2 - \bar{d}_i + p_i$. Consequently, around 0, $\Theta(h) = c_i p_i$.

□

Proposition 61. *Let $(t_1, t_2) \in \mathbb{R}^2$ such that $t_1 < t_2$.*

- *If $t_1 \notin O_1$ and $t_2 \notin O_2$, $h \to S_{Sh}(t_1 + h, t_2 - h)$ is linear around 0.*
- *If $t_2 \notin O_2 \cup O(t_1)$, $h \to S_{Sh}(t_1, t_2 - h)$ is linear around 0.*
- *If $t_1 \notin O_1 \cup O(t_2)$, $h \to S_{Sh}(t_1 + h, t_2)$ is linear around 0.*

Proof. We prove the first item. Since $t_1 \notin O_1$ and $t_2 \notin O_2$, $\forall i, h \to W_{Sh}(A_i, t_1 + h, t_2 - h)$ is linear around 0 (Proposition 59). Thus, $h \to S_{Sh}(t_1 + h, t_2 - h)$ is linear around 0. The same proof applies for other items (Proposition 60 and its symmetric counterpart are used). □

Proof of Proposition 58. The implication from left to right is obvious. Suppose now that the right hand side of the equivalence holds and that there exists an interval $[t_1, t_2)$ for which the slack is strictly negative. As in the partially elastic case, we remark that when t_1 is smaller than $r_{\min} = \min(r_i)$, the slack strictly increases when t_1 decreases. Similarly, when t_2 is greater than $\bar{d}_{\max} = \max(\bar{d}_i)$, the slack strictly increases when t_2 increases. Since the slack function is also continuous, it assumes a minimal value over an interval $[t_1, t_2)$ with $r_{\min} \leq t_1 \leq t_2 \leq \bar{d}_{\max}$. Let us consequently select a pair (t_1, t_2) at which the slack is minimal. In case several pairs (t_1, t_2) minimize the slack, an interval with maximal length is chosen. Since this slack is strictly negative, we must have $t_1 < t_2$.

- **Case 1:** If $t_1 \notin O_1$ and $t_2 \notin O_2$, then according to Proposition 61, the function $q(h) = S_{Sh}(t_1 + h, t_2 - h)$ is linear around 0. Since (t_1, t_2) is a global minimum of the slack, $q(h)$ is constant around 0, which contradicts the fact that the length of $[t_1, t_2)$ is maximal.
- **Case 2:** If $t_1 \in O_1$ then $t_2 \notin O_2 \cup O(t_1)$, otherwise the slack is non-negative. According to Proposition 61, $q(h) = S_{Sh}(t_1, t_2 - h)$ is linear around 0. Since (t_1, t_2) is a global minimum of the slack, $q(h)$ is constant around 0, which contradicts the fact that the length of $[t_1, t_2)$ is maximal.
- **Case 3:** If $t_2 \in O_2$ then $t_1 \notin O_1 \cup O(t_2)$, otherwise the slack is non-negative. According to Proposition 61, $q(h) = S_{Sh}(t_1 + h, t_2)$ is linear around 0. Since (t_1, t_2) is a global minimum of the slack, $q(h)$ is constant around 0, which contradicts the fact that the length of $[t_1, t_2)$ is maximal.

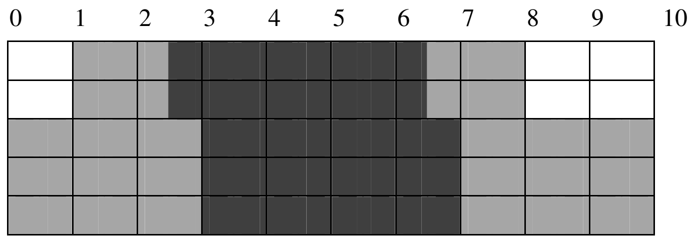
The combination of cases 1, 2 and 3 leads to a contradiction. □

Proposition 58 provides a characterization of interesting intervals over which the slack must be computed to ensure it is always non-negative over any interval. This characterization is weaker than the one proposed for the partially elastic case where the interesting time intervals $[t_1, t_2)$ are in the Cartesian product of the set of the earliest start times and of the set of latest end times. However, there are still only $O(n^2)$ relevant pairs (t_1, t_2) to consider. Some of these pairs belong to the Cartesian product $O_1 * O_2$. The example of Figure 9.6 proves that some pairs do not. In this example, (resource of capacity 2 and 5 activities A_1, A_2, A_3, A_4, A_5), the pair (1, 9) corresponds to the minimal slack and does not belong to $\{0, 1, 4, 5, 6\} * \{4, 5, 6, 8, 10\}$. In this interval, the slack is negative, which proves that there is no feasible schedule. Notice that neither the fully elastic nor the partially elastic relaxation can trigger a contradiction.

Computing Energy in Quadratic time

We propose an $O(n^2)$ algorithm to compute the required energy consumption W_{Sh} over all interesting pairs of time points. Actually, the algorithm first computes all the relevant values taken by W_{Sh} over time intervals $[t_1, t_2)$ with $t_1 \in O_1$, and then computes all the relevant values taken by W_{Sh} over time intervals $[t_1, t_2)$ with $t_2 \in O_2$. The characterization obtained in the previous section ensures

	i	$\bar{d}_i$	P_i	c_i
A_1	1	8	4	1
A_2	1	8	4	1
A_3	0	10	4	1
A_4	0	10	4	1
A_5	0	10	4	1



$t_2 \setminus t_1$	0	1	2
8	16 – 4 * 2 - 2 * 3 = 2	14 - 4 * 2 - 2 * 3 = 0	12 - 3 * 2 - 2 * 3 = 0
9	18 – 4 * 2 - 3 * 3 = 1	16 - 4 * 2 - 3 * 3 = -1	14 - 3 * 2 - 2 * 3 = 2
10	20 – 4 * 2 - 4 * 3 = 0	18 - 4 * 2 - 3 * 3 = 1	16 - 3 * 2 - 2 * 3 = 4

Figure 9.6: Some relevant time intervals are outside $O_1 * O_2$

that all the interesting time intervals are examined in at least one of these steps. For symmetry reasons, we will only describe the first computation. It is achieved by the same type of technique than in the partially elastic case. An outer loop iterates on all values $t_1 \in O_1$ sorted in increasing order. Then, we consider the function $t \rightarrow W_{Sh}(A_i, t_1, t)$. This function is linear on the intervals delimited by the values $\bar{d}_i, r_i + p_i, \bar{d}_i - p_i$ and $r_i + \bar{d}_i - t_1$. We rely on this property to incrementally maintain the slope of the function $t \rightarrow W_{Sh}(A_i, t_1, t)$.

There are some few differences with the partially elastic case.

- The computation of t_2 is slightly more complex since there are more interesting values to consider.
- One does not need to reorder any list: when t_1 increases, none of the values $\bar{d}_i, r_i + p_i, \bar{d}_i - p_i$ and $r_i + \bar{d}_i - t_1$ changes except the last one; which corresponds to the list *Middle*. Since *Middle* is initially sorted in increasing order of $r_i + \bar{d}_i$, it is also sorted in increasing order of $r_i + \bar{d}_i - t_1$.
- In line 43, one shall in fact be careful not to update the slope more than once for the same tuple (t_1, t_2, i). This can be done easily by marking the activity A_i with t_2. We have not included this marking in the pseudo-code to keep it simple.

Time-Bound Adjustments

As for partially elastic adjustments, the values of W_{Sh} can be used to adjust time bounds. Given an activity A_i and a time interval $[t_1, t_2)$ with $t_2 < \bar{d}_i$, we examine whether A_i can end before t_2.

Proposition 62. *If there is a time interval* $[t_1, t_2)$ *such that*

$$W_{Sh}(t_1, t_2) - W_{Sh}(A_i, t_1, t_2) + c_i p_i^+(t_1) > C(t_2 - t_1)$$

then a valid lower bound of the end time of A_i *is:*

$$t_2 + \frac{1}{c_i}(W_{Sh}(t_1, t_2) - W_{Sh}(A_i, t_1, t_2) + c_i p_i^+(t_1) - C(t_2 - t_1))$$

Proof. Similar to proof of Proposition 53: A_i cannot be finished before t_2 and at least

$$W_{Sh}(t_1, t_2) - W_{Sh}(A_i, t_1, t_2) + c_i p_i^+(t_1) - C(t_2 - t_1)$$

units of energy must come after t_2. □

Algorithm 15 Computation of $W_{Sh}(t_1, t_2)$

```
1: $D :=$ activities sorted in increasing order of $\bar{d}_i$
2: $Eet :=$ activities sorted in increasing order of $r_i + p_i$
3: $Lst :=$ activities sorted in increasing order of $\bar{d}_i - p_i$
4: $Middle :=$ activities sorted in increasing order of $r_i + \bar{d}_i$
5: for $t_1$ in the set $O_1$ (sorted in increasing order) do
6:   $i^D := 0, i^{Eet} := 0, i^{Lst} := 0, i^{Middle} := 0$
7:   $W := 0, t_2^{\text{old}} := t_1, slope := \sum_i W_{Sh}(i, t_1, t_1 + 1)$
8:   while $i^D < n$ or $i^{Eet} < n$ or $i^{Lst} < n$ or $i^{Middle} < n$ do
9:     if $i^D < n$ then
10:      $t_2^D := \bar{d}_{D[i^D+1]}$
11:    else
12:      $t_2^D := \infty$
13:    end if
14:    if $i^{Eet} < n$ then
15:      $t_2^{Eet} := r_{Eet[i^{Eet}+1]} + p_{Eet[i^{Eet}+1]}$
16:    else
17:      $t_2^{Eet} := \infty$
18:    end if
19:    if $i^{Lst} < n$ then
20:      $t_2^{Lst} := \bar{d}_{Lst[i^{Lst}+1]} - p_{Lst[i^{Lst}+1]}$
21:    else
22:      $t_2^{Lst} := \infty$
23:    end if
24:    if $i^{Middle} < n$ then
25:      $t_2^{Middle} := r_{Middle[i^{Middle}+1]} + \bar{d}_{Middle[i^{Middle}+1]} - t_1$
26:    else
27:      $t_2^{Middle} := \infty$
28:    end if
29:    $t_2 := \min(t_2^D, t_2^{Eet}, t_2^{Lst}, t_2^{Middle})$
30:    if $t_2 = t_2^D$ then
31:      $i^D := i^D + 1, i := D[i^D]$
32:    else if $t_2 = t_2^{Eet}$ then
33:      $i^{Eet} := i^{Eet} + 1, i := Eet[i^{Eet}]$
34:    else if $t_2 = t_2^{Lst}$ then
35:      $i^{Lst} := i^{Lst} + 1, i := Lst[i^{Lst}]$
36:    else if $t_2 = t_2^{Middle}$ then
37:      $i^{Middle} := i^{Middle} + 1, i := Middle[i^{Middle}]$
38:    end if
39:    if $t_1 < t_2$ then
40:      $W := W + slope(t_2 - t_2^{\text{old}})$
41:      $W_{Sh}(t_1, t_2) := W$ (energy over $[t_1, t_2)$)
42:      $t_2^{\text{old}} := t_2$
43:      $slope := slope + W_{Sh}(i, t_1, t_2 + 1) - 2W_{Sh}(i, t_1, t_2) + W_{Sh}(i, t_1, t_2 - 1)$
44:    end if
45:  end while
46: end for
```

Similarly, when

$$W_{Sh}(t_1,t_2) - W_{Sh}(A_i,t_1,t_2) + c_i \min(t_2 - t_1, p_i^+(t_1)) > C(t_2 - t_1),$$

A_i cannot start before t_1 and a valid lower bound of the start time of A_i is

$$t_2 - \frac{1}{c_i}(C(t_2 - t_1) - W_{Sh}(t_1,t_2) + W_{Sh}(A_i,t_1,t_2)).$$

There is an obvious $O(n^3)$ algorithm to compute all the adjustments corresponding to Proposition 62 which can be obtained on the intervals $[t_1, t_2)$ which correspond to potential local minima of the slack function. There are $O(n^2)$ intervals of interest and n activities which can be adjusted. Given an interval and an activity, the adjustment procedure runs in $O(1)$. The overall complexity of the algorithm is then $O(n^3)$. An interesting open question is whether there is a quadratic algorithm to compute all the adjustments on the $O(n^2)$ intervals under consideration.

Algorithm 16 An algorithm for Left-Shift / Right-Shift adjustments

```
 1: for all relevant time-intervals $[t_1, t_2)$ (cf., Proposition 58) do
 2:    $W := 0$
 3:    for $i \in \{1, \dots, n\}$ do
 4:       $W := W + c_i \min(t_2 - t_1, p_i^+(t_1), p_i^-(t_2))$
 5:    end for
 6:    if $W > C(t_2 - t_1)$ then
 7:       there is no feasible schedule, exit
 8:    else
 9:       for $i \in \{1, \dots, n\}$ do
10:          $SL := C(t_2 - t_1) - W + c_i \min(t_2 - t_1, p_i^+(t_1), p_i^-(t_2))$
11:          if $SL < c_i p_i^+(t_1)$ then
12:             $eet_i := \max(eet_i, t_2 + \lceil (c_i p_i^+(t_1) - SL)/c_i \rceil)$
13:          end if
14:          if $SL < c_i p_i^-(t_2)$ then
15:             $lst_i := \min(lst_i, t_1 - \lceil (c_i p_i^-(t_2) - SL)/c_i \rceil)$
16:          end if
17:       end for
18:    end if
19: end for
```

Note that these Left-Shift / Right-Shift adjustments are more powerful than the adjustments based on partially elastic schedules but they are also more expensive and they only apply to "non-elastic non-preemptive problems." Note also that the satisfiability tests are such that they provide the same answers when an activity A_i, which requires c_i units of the resource, is replaced by c_i activities A_{ij}, each of which requires one unit of the resource. This is not true for the time-bound adjustments. Several questions are still open at this point.

- First, for the Left-Shift / Right-Shift technique, we have shown that the energetic tests can be limited to $O(n^2)$ time intervals. We have also provided a precise characterization of these intervals. However, it could be that this characterization can be sharpened in order to eliminate some intervals and reduce the practical complexity of the corresponding algorithm.

- Second, it seems reasonable to think that our time-bound adjustments could be sharpened. Even though the energetic tests can be limited (without any loss) to a given set of intervals, it could be that the corresponding adjustment rules cannot.

Chapter 10

Comparison of Propagation Techniques

Many deductive rules and algorithms have been presented. The aim of this chapter is to compare the deductive power of these rules and algorithms, thereby providing a much clearer overview of the state of the art in the field.

10.1 Constraint Propagation Rules

The first step to allow a comparison of different constraint propagation techniques is to describe these techniques under a common, precise and unambiguous formalism. Here, most of the resource constraint propagation techniques from the preceding chapters are recalled as rules to be applied until quiescence is reached. Some rules state that arc-consistency or arc-B-consistency must be maintained on a particular mathematical formula. Others are stated as a "condition $\Rightarrow$ conclusion" pair, meaning that the conclusion must be deduced whenever the condition is verified.

Each rule is identified with a two-part name XX-*, where the first part XX identifies the most general case in which the rule applies. Five cases are distinguished: FE for Fully Elastic, CP for Cumulative Preemptive, CNP for Cumulative Non-Preemptive, 1P for One-Machine Preemptive, and 1NP for One-Machine Non-Preemptive.

In the **fully elastic case (FE)**, the amount of resource R assigned to an activity A_i can, at any time t, pick any value $E(A_i, t, R)$ between 0 and the resource capacity $cap(R)$, provided that the sum over time of the assigned capacity $E(A_i, t, R)$ equals a given amount of energy $E(A_i, R)$. For each activity A_i, the following constraints apply:

$$\begin{aligned}
&start(A_i) = \min_{t \in set(A_i)} (t) \\
&end(A_i) = \max_{t \in set(A_i)} (t+1) \\
&proc(A_i) = |set(A_i)| \\
&[X(A_i, t) = 1] \Leftrightarrow [t \in set(A_i)] \\
&[E(A_i, t, R) \neq 0] \Leftrightarrow [X(A_i, t) \neq 0] \\
&\sum_t E(A_i, t, R) = E(A_i, R) \\
&\sum_i E(A_i, t, R) \leq cap(R)
\end{aligned}$$

The fully elastic case is interesting because the constraint propagation techniques that are developed for this case can be applied to any fully elastic, partially elastic, or non-elastic scheduling problem (because it is the less constrained).

In the **cumulative preemptive case (CP)**, an additional constraint specifies that, for each activity A_i, the amount of resource $cap(A_i, R)$ assigned to A_i whenever A_i executes does not vary and :

$$E(A_i, t, R) = cap(A_i, R) X(A_i, t)$$

The **One-Machine preemptive case (1P)** is the restriction of the CP case to a resource of capacity 1, *i.e.*, $cap(A_i, R) = cap(R) = 1$.

Finally, the **cumulative non-preemptive case (CNP)** and the **One-Machine non-preemptive case (1NP)** are obtained from CP and 1P by adding, for each activity A_i, a constraint which specifies that A_i cannot be interrupted:

$$set(A_i) = [start(A_i), end(A_i))$$

As already mentioned, constraint propagation rules developed for the FE case can be applied in the CP, 1P, CNP and 1NP cases. Similarly, constraint propagation rules developed for the CP case can be applied in the 1P, CNP and 1NP cases. In the 1NP cases, all the rules, including those developed for the 1P and CNP cases, can be applied.

To keep things simple, it is in all cases assumed that, for every activity A_i, the processing time $proc(A_i)$ and the energy requirement $E(A_i, R)$ are constrained to be strictly positive. We recall some notations:

- p_i is the processing time of activity A_i, *i.e.*, the value of $proc(A_i)$ when $proc(A_i)$ is bound. When $proc(A_i)$ is not bound, constraint propagation rules can usually be applied with p_i equal to the minimal value in the domain of $proc(A_i)$.
- $c_{i,R}$ is the capacity (of the resource R under consideration) required by activity A_i, *i.e.*, the value of $cap(A_i, R)$ when $cap(A_i, R)$ is bound. When $cap(A_i, R)$ is not bound, constraint propagation rules can usually be applied with $c_{i,R}$ equal to the minimal value in the domain of $cap(A_i, R)$.
- $e_{i,R}$ is the energy (of the resource R under consideration) required by activity A_i, *i.e.*, the value of $E(A_i, R)$ when $E(A_i, R)$ is bound. When $E(A_i, R)$ is not bound, constraint propagation rules can usually be applied with $e_{i,R}$ equal to the minimal value in the domain of $E(A_i, R)$.
- r_Ω is the release date of the set of activities Ω. $r_\Omega = \min_{A_i \in \Omega} r_i$.
- $\bar{d}_\Omega$ is the deadline of the set of activities Ω. $\bar{d}_\Omega = \max_{A_i \in \Omega} \bar{d}_i$.
- $eetmin_\Omega$ is the smallest value in $\{eet_i : A_i \in \Omega\}$.
- $lstmax_\Omega$ is the largest value in $\{lst_i : A_i \in \Omega\}$.
- p_Ω is the processing time of the set of activities Ω. $p_\Omega = \sum_{A_i \in \Omega} p_i$.
- e_Ω is the energy required by the set of activities Ω. $e_\Omega = \sum_{A_i \in \Omega} e_i$.

A constraint propagation rule $\mathcal{R}$ is said to be "dominated" by another constraint propagation rule $\mathcal{R}'$ (or by a set of constraint propagation rules $\mathcal{S}$) if and only if all the deductions made by $\mathcal{R}$ are also made by $\mathcal{R}'$ (respectively, $\mathcal{S}$). Let us note that a dominated constraint propagation rule $\mathcal{R}$ can still prove useful in practice, for example when the rule $\mathcal{R}'$ that dominates $\mathcal{R}$ is too costly to be applied in a systematic fashion. Yet, knowing that $\mathcal{R}$ is dominated by $\mathcal{R}'$ can be very useful, especially when developing a search algorithm based on constraint propagation results. Indeed, we know in this case that the search tree obtained when $\mathcal{R}$ and $\mathcal{R}'$ are both applied, and the search tree obtained when $\mathcal{R}'$ alone is applied, are identical. In other terms, applying $\mathcal{R}$ when $\mathcal{R}'$ is already applied will not save any search when $\mathcal{R}$ is dominated by $\mathcal{R}'$. Conversely, knowing that two rules $\mathcal{R}$ and $\mathcal{R}'$ do not dominate each other suggests that the $(\mathcal{R} + \mathcal{R}')$ combination might perform better than each of the two rules considered separately.

Let us note that, in some scheduling problems, resource capacity varies over time (*e.g.*, when less people work at night than during the day). When only $c < cap(R)$ resource units are available over an interval $[t_1, t_2)$, a "fake" activity can be introduced and said to require capacity $cap(R) - c$ between t_1 and t_2. This transformation will have an impact on the deductive power of some constraint propagation techniques (but not all). However, the dominance relations remain the same.

In the following, we do not consider the possible interactions between resources R so, to simplify notation, the index R will be omitted. C, c_i, e_i, p_i denote the capacity of the resource, the capacity required by activity A_i, the energy required by activity A_i and the processing time of activity A_i. To keep the presentation simple, we assume that the variables $cap(R)$, $cap(A_i, R)$, $E(A_i, R)$ and $proc(A_i)$ are bound to these values. Depending on the context, it shall be obvious whether C, c_i, e_i and p_i can be replaced by the lower or the upper bound of the corresponding variables.

10.1.1 Time-Table Constraints

FE-TT, CP-TT and CNP-TT: Time-Tables in the fully elastic, preemptive and non-preemptive cases

This constraint propagation technique consists of maintaining arc-B-consistency on the constraint $\sum_i E(A_i, t) \leq C$ and using the maximal values $ub(E(A_i, t))$ of $E(A_i, t)$ to compute the earliest and latest start and end times of activities:

$$\begin{aligned}
start(A_i) &\geq \min\{t : ub(E(A_i, t)) > 0\} \\
end(A_i) &\geq \min\{t : \sum_{u<t} ub(E(A_i, u)) \geq e_i\} \\
start(A_i) &\leq \max\{t : \sum_{u \geq t} ub(E(A_i, u)) \geq e_i\} \\
end(A_i) &\leq \max\{t : ub(E(A_i, t)) > 0\} + 1
\end{aligned}$$

The FE, CP and CNP cases must be distinguished because the relations between $E(A_i, t)$ and the start and end time variables differ. In the non-preemptive case, $E(A_i, t)$ is equal to c_i for every t in $[lst_i, eet_i)$. In the preemptive case, either the user or a search procedure must explicitly decide at which time points each activity executes. However, when $start(A_i)$, $proc(A_i)$ and $end(A_i)$ are bound to values such that $start(A_i) + proc(A_i) = end(A_i)$, $E(A_i, t)$ is equal to c_i for every t in $[start(A_i), end(A_i))$. In the fully elastic case, the relation is even weaker because, even when t is known to belong to $set(A_i)$, $E(A_i, t)$ can take any value between 1 and C.

10.1.2 Disjunctive Constraints

Disjunctive constraints deal with cases in which two activities cannot overlap in time. Such a situation is of course common in One-Machine scheduling, but also occurs in cumulative scheduling, when the sum of the capacities required by two activities exceeds the capacity of the resource. Let us note that the following rule, CNP-DISJ, can a priori be generalized to triples, quadruples, *etc.*, of activities, but with a significant increase in the number of constraints and in the number of disjuncts per constraint. In practice, disjunctive constraints are used only for pairs of activities.

CNP-DISJ: Disjunctive constraints in the non-preemptive case

Let A_i and A_j be two activities with $c_i + c_j > C$. In the non-preemptive case, the disjunctive constraint propagation technique consists of maintaining arc-B-consistency on the formula:

$$[end(A_i) \leq start(A_j)] \vee [end(A_j) \leq start(A_i)]$$

In the 1NP case, all the pairs of activities that require the same resource are related by such a disjunction. This is the origin of the term "disjunctive scheduling."

CP-DISJ: Disjunctive constraints in the preemptive case

Let A_i and A_j be two activities with $c_i + c_j > C$. In the preemptive case, the disjunctive constraint propagation technique consists of maintaining arc-B-consistency on the formula:

$$\begin{aligned}
& [start(A_i) + proc(A_i) + proc(A_j) \leq end(A_i)] \\
\vee\ & [start(A_i) + proc(A_i) + proc(A_j) \leq end(A_j)] \\
\vee\ & [start(A_j) + proc(A_i) + proc(A_j) \leq end(A_i)] \\
\vee\ & [start(A_j) + proc(A_i) + proc(A_j) \leq end(A_j)]
\end{aligned}$$

10.1.3 Edge-Finding

"Edge-Finding" constraint propagation techniques reason about the order in which activities execute on a given resource. In the 1NP case, Edge-Finding consists of determining whether a given activity A_i must execute before (or after) a given set of activities Ω. Cumulative and preemptive cases are more complex since several activities can overlap (on a cumulative resource) or preempt one another. Then Edge-Finding consists of determining whether an activity A_i must start or end before (or after) a set of activities Ω.

1NP-EF: Edge-Finding in the One-Machine non-preemptive case

Let $A_i \ll A_j$ ($A_i \gg A_j$) mean that A_i is before (after) A_j and $A_i \ll \Omega$ ($A_i \gg \Omega$) mean that A_i is before (after) all the activities in Ω. In the One-Machine non-preemptive case, the Edge-Finding technique consists in applying the following rules, and their symmetric counterparts:

$$[A_i \notin \Omega] \wedge \left[\bar{d}_\Omega - r_{\Omega \cup \{A_i\}} < p_{\Omega \cup \{A_i\}}\right] \Rightarrow [A_i \gg \Omega]$$
$$[A_i \gg \Omega] \Rightarrow \left[\mathit{start}(A_i) \geq \max_{\emptyset \neq \Omega' \subseteq \Omega} (r_{\Omega'} + p_{\Omega'})\right]$$

1P-EF: Edge-Finding in the One-Machine preemptive case

In the preemptive case, the Edge-Finding rules no longer order activities but start and end times of activities. If $A_i \succ \Omega$ means "A_i ends after all activities in Ω", then the following rules (and their symmetric counterparts) are obtained:

$$[A_i \notin \Omega] \wedge \left[\bar{d}_\Omega - r_{\Omega \cup \{A_i\}} < p_{\Omega \cup \{A_i\}}\right] \Rightarrow [A_i \succ \Omega]$$
$$[A_i \succ \Omega] \Rightarrow \left[\mathit{end}(A_i) \geq \max_{\Omega' \subseteq \Omega} (r_{\Omega' \cup \{A_i\}} + p_{\Omega' \cup \{A_i\}})\right]$$
$$[A_i \succ \Omega] \wedge \left[\bar{d}_\Omega - r_\Omega = p_\Omega\right] \wedge [r_\Omega \leq r_i] \Rightarrow \left[\mathit{start}(A_i) \geq \bar{d}_\Omega\right]$$

In the "mixed" case with both interruptible and non-interruptible activities, the 1NP-EF rules can be applied whenever A_i is not interruptible, even if the activities in Ω are interruptible (*cf.*, Section 8.2). On the other hand, if A_i is interruptible, the 1NP-EF rules are not valid and the (weaker) 1P-EF rules must be applied.

FE-EF: Edge-Finding in the fully elastic case

The following rules (and their symmetric counterparts) can be used in the fully elastic case:

$$[A_i \notin \Omega] \wedge \left[C(\bar{d}_\Omega - r_{\Omega \cup \{A_i\}}) < e_{\Omega \cup \{A_i\}}\right] \Rightarrow [A_i \succ \Omega]$$
$$[A_i \succ \Omega] \Rightarrow \left[\mathit{end}(A_i) \geq \max_{\Omega' \subseteq \Omega} \frac{C r_{\Omega' \cup \{A_i\}} + e_{\Omega' \cup \{A_i\}}}{C}\right]$$
$$[A_i \succ \Omega] \wedge \left[C(\bar{d}_\Omega - r_\Omega) = e_\Omega\right] \wedge [r_\Omega \leq r_i] \Rightarrow \left[\mathit{start}(A_i) \geq \bar{d}_\Omega\right]$$

Note that when applied in the CP and in the CNP cases, these rules are sensitive to the use of fake activities to represent intervals during which the resource capacity is c with $0 < c < C$ (because the fake activities are considered as fully elastic).

CNP-EF1: Edge-Finding in the cumulative non-preemptive case

Let $\mathit{rest}(\Omega, c_i) = e_\Omega - (C - c_i)(\bar{d}_\Omega - r_\Omega)$. The following rules (and their symmetric counterparts) summarize Section 9.3.3.

$$[A_i \notin \Omega] \wedge \left[C(\bar{d}_\Omega - r_{\Omega \cup \{A_i\}}) < e_{\Omega \cup \{A_i\}}\right] \Rightarrow [A_i \succ \Omega]$$
$$[A_i \succ \Omega] \wedge [\emptyset \neq \Omega' \subseteq \Omega] \wedge [\mathit{rest}(\Omega', c_i) > 0]$$
$$\Rightarrow \left[\mathit{start}(A_i) \geq r_{\Omega'} + \frac{\mathit{rest}(\Omega', c_i)}{c_i}\right]$$

CNP-EF2: Edge-Finding in the cumulative non-preemptive case

The following rules (and their symmetric counterparts) summarize Section 9.3.4.

$$\begin{aligned}&[A_i \notin \Omega] \wedge [r_i \leq r_\Omega < eet_i] \wedge \left[C(\bar{d}_\Omega - r_\Omega) < e_\Omega + c_i(eet_i - r_\Omega)\right]\\&\Rightarrow [A_i \succ \Omega]\\&[A_i \succ \Omega] \wedge [\emptyset \neq \Omega' \subseteq \Omega] \wedge [rest(\Omega', c_i) > 0]\\&\Rightarrow \left[start(A_i) \geq r_{\Omega'} + \frac{rest(\Omega', c_i)}{c_i}\right]\end{aligned}$$

10.1.4 Not-First, Not-Last

"Not-First" and "Not-Last" rules are counterparts to Edge-Finding rules. They deduce that an activity A_i cannot be the first (or the last) to execute in $\Omega \cup \{A_i\}$.

1NP-NFNL: One-Machine "Not-First" and "Not-Last" rules

The following rule (and its symmetric counterpart) can be applied in the One-Machine non-preemptive case:

$$[A_i \notin \Omega] \wedge \left[\bar{d}_\Omega - r_i < p_{\Omega \cup \{A_i\}}\right] \Rightarrow [start(A_i) \geq eetmin_\Omega]$$

CNP-NFNL: Cumulative "Not-First" and "Not-Last" rules

The following rule (and its symmetric counterparts) summarize Section 9.3.5.

$$\begin{aligned}&[A_i \notin \Omega] \wedge [r_\Omega \leq r_i < eetmin_\Omega] \wedge\\&\left[C(\bar{d}_\Omega - r_\Omega) < e_\Omega + c_i(\min(eet_i, \bar{d}_\Omega) - r_\Omega)\right]\\&\Rightarrow [start(A_i) \geq eetmin_\Omega)]\end{aligned}$$

10.1.5 Energetic Reasoning

The Edge-Finding and the "Not-First" and "Not-Last" rules are such that the activities that do not belong to Ω do not contribute to the analysis of the resource usage between $r_{\Omega \cup \{A_i\}}$ and $\bar{d}_{\Omega \cup \{A_i\}}$. On the contrary, energetic reasoning rules compute the minimal contribution $W(A_j, t_1, t_2)$ of each activity A_j to a given interval $[t_1, t_2)$.

In the preemptive case, the required energy consumption of A_j over $[t_1, t_2)$ is evaluated as follows:

$$W_{PE}(A_j, t_1, t_2) = c_j \max(0, p_j - \max(0, t_1 - r_j) - \max(0, \bar{d}_j - t_2))$$

The notation W_{PE} introduced in Section 9.2.3 refers to the fact that this value corresponds to a particular relaxation of the cumulative resource constraint, identified as the "Partially Elastic" relaxation. In the non-preemptive case, a stronger value can be used:

$$W_{Sh}(A_j, t_1, t_2) = c_j \min(t_2 - t_1, p_j^+(t_1), p_j^-(t_2))$$

with $p_j^+(t_1) = \max(0, p_j - \max(0, t_1 - r_j))$ and $p_j^-(t_2) = \max(0, p_j - \max(0, \bar{d}_j - t_2))$. The notation W_{Sh} refers to the fact that the best value is obtained by shifting the activity either to the left (*i.e.*, to its earliest possible execution interval) or to the right (to its latest possible execution interval).

Many rules have been developed based on energetic reasoning (see, for example [19, 23, 29, 97, 159]). Only the most significant rules are recalled below. These rules are in practice applied only to some intervals $[t_1, t_2)$ (*cf.*, Section 9.3.6). For the theoretical comparison with other rules, it will be assumed that all intervals $[t_1, t_2)$ are considered.

CP-ER: Energetic reasoning in the preemptive case

Let $d^+(A_i, t_1, t_2) = \sum_{A_j \neq A_i} W_{PE}(A_j, t_1, t_2) + c_i p_i^+(t_1) - C(t_2 - t_1)$ and

$d^-(A_i,t_1,t_2) = \sum_{A_j \neq A_i} W_{PE}(A_j,t_1,t_2) + c_i p_i^-(t_2) - C(t_2-t_1)$. The following rules can be applied:

$$\left[\sum_{A_j} W_{PE}(A_j,t_1,t_2) - C(t_2-t_1) > 0\right] \Rightarrow \textit{contradiction}$$

$$\left[d^+(A_i,t_1,t_2) > 0\right] \Rightarrow \left[\textit{end}(A_i) \geq t_2 + \frac{d^+(A_i,t_1,t_2)}{c_i}\right]$$

$$[t_1 \leq r_i] \wedge \left[\sum_{A_j \neq A_i} W_{PE}(A_j,t_1,t_2) = C(t_2-t_1)\right] \Rightarrow [\textit{start}(A_i) \geq t_2]$$

$$\left[d^-(A_i,t_1,t_2) > 0\right] \Rightarrow \left[\textit{start}(A_i) \leq t_1 - \frac{d^-(A_i,t_1,t_2)}{c_i}\right]$$

$$\left[\bar{d}_i \leq t_2\right] \wedge \left[\sum_{A_j \neq A_i} W_{PE}(A_j,t_1,t_2) = C(t_2-t_1)\right] \Rightarrow [\textit{end}(A_i) \leq t_1]$$

CNP-ER: Energetic reasoning in the non-preemptive case

Let $d^+(A_i,t_1,t_2) = \sum_{A_j \neq A_i} W_{Sh}(A_j,t_1,t_2) + c_i p_i^+(t_1) - C(t_2-t_1)$ and $d^-(A_i,t_1,t_2) = \sum_{A_j \neq A_i} W_{Sh}(A_j,t_1,t_2) + c_i p_i^-(t_2) - C(t_2-t_1)$. The following rules can be applied:

$$\left[\sum_{A_j} W_{Sh}(A_j,t_1,t_2) - C(t_2-t_1) > 0\right] \Rightarrow \textit{contradiction}$$

$$\left[d^+(A_i,t_1,t_2) > 0\right] \Rightarrow \left[\textit{end}(A_i) \geq t_2 + \frac{d^+(A_i,t_1,t_2)}{c_i}\right]$$

$$[t_1 \leq r_i] \wedge \left[\sum_{A_j \neq A_i} W_{Sh}(A_j,t_1,t_2) = C(t_2-t_1)\right] \Rightarrow [\textit{start}(A_i) \geq t_2]$$

$$\left[d^-(A_i,t_1,t_2) > 0\right] \Rightarrow \left[\textit{start}(A_i) \leq t_1 - \frac{d^-(A_i,t_1,t_2)}{c_i}\right]$$

$$\left[\bar{d}_i \leq t_2\right] \wedge \left[\sum_{A_j \neq A_i} W_{Sh}(A_j,t_1,t_2) = C(t_2-t_1)\right] \Rightarrow [\textit{end}(A_i) \leq t_1]$$

CNP-ER-DISJ: Energetic reasoning in the non-preemptive case

$$\begin{aligned}
&[t_1 = r_i] \wedge \left[t_2 = \bar{d}_j\right] \wedge [t_1 < t_2] \wedge [c_i + c_j > C] \\
&\wedge \left[\sum_{A_k \notin \{A_i,A_j\}} W_{Sh}(A_k,t_1,t_2) + c_i p_i + c_j p_j > C(t_2-t_1)\right] \\
&\Rightarrow [\textit{end}(A_j) \leq \textit{start}(A_i)]
\end{aligned}$$

10.2 Dominance Relations

This section summarizes the dominance relations under the following assumptions:

- The study considers only the deduction of new time-bounds (*i.e.*, earliest and latest start and end times) from given time-bounds. When Time-Tables are used, other deductions are in fact performed, since the $E(A_i,t)$ variables are updated. When several constraints refer to the same $E(A_i,t)$ variables, this may lead to extra constraint propagation.

- In the CP, 1P, CNP and 1NP cases, we consider only the case in which the processing time and the resource capacity requirement of each activity are fixed. In the non-preemptive case, this assumption is particularly important because it implies that $start(A_i) + p_i = end(A_i)$; hence any deduction concerning $start(A_i)$ is equivalent to a deduction concerning $end(A_i)$, and conversely.

- In the preemptive case, a peculiar "special effect" occurs when the $start(A_i)$ variable is bound. Indeed, the value of this variable is then known to belong to $set(A_i)$, an information which could theoretically be used in some constraint propagation rules (*i.e.*, those based on Time-Tables). A similar effect occurs when the $end(A_i)$ variable is bound. However, it appears that the use of this additional information would greatly impact the CPU time and memory space needed to update and exploit the Time-Tables. So, in practice, these "special effects" are ignored. To remain consistent with the actual practice, we also ignore these effects in the following comparisons.

- Extensions of the rules (*e.g.*, to consider also setup times between activities [51, 103]) are ignored. Hence, some dominance relations may no longer hold when such extensions are considered.

Let us remark that the conditions of the CNP-EF2, CNP-NFNL and CNP-ER-DISJ rules are not monotonic (while FE-TT, CP-TT, CNP-TT, CP-DISJ, CNP-DISJ, FE-EF, 1P-EF, 1NP-EF, CNP-EF1, 1NP-NFNL, CP-ER and CNP-ER have monotonic conditions). This means that better time-bounds for some activities may prevent the application of the rule (and hence the drawing of the corresponding conclusion). For example, consider two non-interruptible activities A_i and A_j with $r_i = X$, $\bar{d}_i = 7$, $r_j = 1$, $\bar{d}_j = 5$, $p_i = 3$, $p_j = 3$, $c_i = 1$, $c_j = 1$ and $C = 1$. This example shows that CNP-EF2 is non-monotonic. Indeed, for $X = 3$, nothing is deduced, while for $X = 0$ or $X = 1$, the rule fires and finds $start(A_i) \geq 4$.

Dominance relations between non-monotonic rules are difficult to establish because the fact that a non-monotonic rule does not apply in a given state does not prove that it could not have been applied in a previous state. However, dominance relations between a monotonic rule $\mathcal{R}$ and a non-monotonic rule $\mathcal{R}'$ can be established without such worry.

Each of the FE, CP, 1P, CNP and 1NP cases is considered in turn. We note $\mathcal{R} \leq \mathcal{R}_1 + \mathcal{R}_2 + \ldots + \mathcal{R}_n$ when the rule $\mathcal{R}$ is dominated by the conjunction of $\mathcal{R}_1$, $\mathcal{R}_2$, …, $\mathcal{R}_n$ and $\mathcal{R} = \mathcal{R}_1 + \mathcal{R}_2 + \ldots + \mathcal{R}_n$ when $\mathcal{R}$ and the conjunction of $\mathcal{R}_1$, $\mathcal{R}_2$, …, $\mathcal{R}_n$ are equivalent. Similarly, we will use the notation $\mathcal{R} \not\leq \mathcal{R}_1 + \mathcal{R}_2 + \ldots + \mathcal{R}_n$ when $\mathcal{R}$ is not dominated by the conjunction of $\mathcal{R}_1$, $\mathcal{R}_2$, …, $\mathcal{R}_n$.

10.2.1 The Fully Elastic Case

Proposition 63. *FE-TT $\leq$ FE-EF*

Proof. It is shown in Proposition 48 (Section 9.1) that FE-EF deduces the best possible bounds in the fully elastic case. Hence, FE-TT can never provide better bounds than FE-EF. □

10.2.2 The Cumulative Preemptive Case

Proposition 64. *FE-TT $\leq$ CP-TT*

Proof. Obvious. □

Proposition 65. *FE-TT $\leq$ FE-EF $\leq$ CP-ER*

Proof. FE-TT $\leq$ FE-EF is proven in Section 10.2.1. Let us now prove FE-EF $\leq$ CP-ER.

FE-EF provides two types of conclusions. Let us first consider a conclusion of the first type, *i.e.*, a new lower bound for $end(A_i)$. It is shown in Section 9.1 that there exists a simple transformation of an instance of the Fully Elastic Problem into an instance of the One-Machine Preemptive Problem such that this conclusion corresponds to a conclusion of the 1P-EF rule on the transformed instance (Transformation 1). In addition, it is shown in [10] that if a set Ω' provides the conclusion, then this rule can be applied with $\Omega = \Omega'$ (to provide the same conclusion). This implies $d^+(A_i, t_1, t_2) > 0$ for $t_1 = r_{\Omega'}$ and $t_2 = \bar{d}_{\Omega'}$. Then the CP-ER energetic rule deduces $end(A_i) \geq t_2 + d^+(A_i, t_1, t_2)/c_i$. Assuming $c_i \leq C$, this cannot be smaller than $t_1 + (e_{\Omega'} + c_i p_i^+(t_1))/C$, which in turn cannot be smaller than $r_{\Omega' \cup \{A_i\}} + e_{\Omega' \cup \{A_i\}}/C$. So the conclusion made by CP-ER is not weaker than the conclusion made by FE-EF.

Let us now consider a conclusion of the second type, *i.e.*, a new lower bound for $start(A_i)$. The CP-ER rule obtains the same conclusion with $t_1 = r_\Omega$ and $t_2 = \bar{d}_\Omega$. Hence FE-EF $\leq$ CP-ER. □

10.2.3 The One-Machine Preemptive Case

Proposition 66.
FE-TT = CP-TT ≤ FE-EF = 1P-EF = CP-ER
CP-DISJ ≤ FE-EF = 1P-EF = CP-ER

Proof. When $C = 1$, any fully elastic schedule is a valid preemptive schedule, and conversely. This implies that FE-TT and CP-TT are equivalent and that FE-EF and 1P-EF are equivalent. In addition, 1P-EF is known to compute the best possible bounds. Hence 1P-EF dominates all the other rules. Since CP-ER dominates FE-EF (= 1P-EF), this implies that CP-ER, FE-EF and 1P-EF are equivalent and dominate all the other rules. □

10.2.4 The Cumulative Non-Preemptive Case

Proposition 67. *FE-TT ≤ FE-EF ≤ CP-ER*

Proof. See Sections 10.2.1 and 10.2.2. □

Proposition 68.
FE-TT ≤ CP-TT ≤ CNP-TT
CP-DISJ ≤ CNP-DISJ
CP-ER ≤ CNP-ER

Proof. Each of these relations is obvious when we assume that the activities have fixed processing times. □

Proposition 69. *FE-EF ≤ CNP-EF1*

Proof. FE-EF provides two types of conclusions. Let us first consider a conclusion of the first type, *i.e.*, a new lower bound for $end(A_i)$. The new lower bound is greater than $r_i + p_i$ and $\bar{d}_\Omega$. It is shown in Transformation 1 that there exists a simple transformation of an instance of the Fully Elastic Problem into an instance of the One-Machine Preemptive Problem such that this conclusion corresponds to a conclusion of the 1P-EF rule on the transformed instance. In addition, it is shown in [10] that if a set Ω' provides the conclusion, then this rule can be applied with $\Omega = \Omega'$ (to provide the same conclusion). Then $rest(\Omega, c_i) > 0$, otherwise $c_i * (\bar{d}_\Omega - r_\Omega)$ units of energy could be allocated to A_i between r_Ω and $\bar{d}_\Omega$, which would not allow FE-EF to update the earliest end time of A_i. This implies that the condition of CNP-EF1 is satisfied for $\Omega = \Omega'$. Finally, the earliest end time of A_i is updated to

$$r_\Omega + \frac{e_\Omega - (C - c_i)(\bar{d}_\Omega - r_\Omega)}{c_i} + p_i$$

Let us show that this conclusion is at least as good as the conclusion raised by FE-EF (provided that c_i does not exceed C). We have:

$$C(\bar{d}_\Omega - r_\Omega) \leq C(\bar{d}_\Omega - r_{\Omega \cup \{A_i\}}) < e_{\Omega \cup \{A_i\}}$$
$$C(C - c_i)(\bar{d}_\Omega - r_\Omega) \leq (C - c_i) e_{\Omega \cup \{A_i\}}$$
$$c_i e_{\Omega \cup \{A_i\}} \leq C e_{\Omega \cup \{A_i\}} - C(C - c_i)(\bar{d}_\Omega - r_\Omega)$$

$$\frac{e_{\Omega\cup\{A_i\}}}{C} \le \frac{e_{\Omega\cup\{A_i\}} - (C-c_i)(\bar{d}_\Omega - r_\Omega)}{c_i}$$
$$\frac{Cr_{\Omega\cup\{A_i\}} + e_{\Omega\cup\{A_i\}}}{C} \le r_\Omega + \frac{e_\Omega - (C-c_i)(\bar{d}_\Omega - r_\Omega)}{c_i} + p_i$$

Let us now consider a conclusion of the second type, *i.e.*, a new lower bound for *start*(A_i). Then CNP-EF1 clearly obtains the same result for $\Omega = \Omega'$. □

Proposition 70. *CNP-EF1* $\le$ *CP-ER*

Proof. Assume CNP-EF1 applies to the pair (A_i, Ω). CP-ER can be applied to A_i, $t_1 = r_{\Omega\cup\{A_i\}}$ and $t_2 = \bar{d}_\Omega$. The result of this step is $end(A_i) \ge t_2 + d^+(A_i, t_1, t_2)/c_i$. This implies that, when CP-ER is applied, eet_i becomes strictly greater than t_2.

Let X denote the earliest start time provided by CNP-EF1 for a given subset Ω'. Let us now apply CP-ER to $t_3 = r_{\Omega'}$ and $t_4 = \bar{d}_{\Omega'}$ as well as the processing time constraint $start(A_i) + p_i = end(A_i)$, until we reach quiescence. Since CP-ER is monotonic, a unique fixpoint can be reached in this manner. At this fixpoint, either $d^+(A_i, t_3, t_4) \le 0$ or $eet_i \ge t_4 + d^+(A_i, t_3, t_4)/c_i$.

This implies $d^+(A_i, t_3, t_4) \le \max(0, c_i(eet_i - t_4))$. Then, since $t_4 \le t_2 < eet_i$, this implies $d^+(A_i, t_3, t_4) \le c_i(eet_i - t_4)$.

However, $d^+(A_i, t_3, t_4) \ge e_{\Omega'} + c_i p_i^+(t_3) - C(t_4 - t_3)$, which implies $d^+(A_i, t_3, t_4) > c_i(X - t_4 - 1 + p_i^+(t_3))$. This implies that at the fixpoint we have $(X - t_4 - 1 + p_i^+(t_3)) < eet_i - t_4$. Hence at the fixpoint $X + p_i^+(t_3) \le eet_i$. Since eet_i is strictly greater than t_3, $p_i^+(t_3) = p_i - \max(0, t_3 - r_i)$. Hence at the fixpoint we have $X - \max(0, t_3 - r_i) \le r_i$, which implies either $X \le r_i$ or $X \le t_3$. But $rest(\Omega', c_i) > 0$ implies $X > t_3$. Consequently, when the fixpoint of CP-ER is reached, the obtained earliest start time r_i is at least as strong as X. □

Proposition 71. *CNP-EF2* $\le$ *CP-ER*

Proof. Assume CNP-EF2 applies to the pair (A_i, Ω). Let $t_1 = r_\Omega$ and $t_2 = \bar{d}_\Omega$. We have $d^+(A_i, t_1, t_2) \ge e_\Omega + c_i p_i^+(t_1) - C(t_2 - t_1)$. Under the conditions of rule CNP-EF2, $p_i^+(t_1) = eet_i - t_1$. Hence, $d^+(A_i, t_1, t_2) > 0$ and CP-ER gives $end(A_i) \ge t_2 + d^+(A_i, t_1, t_2)/c_i$. This implies that, when CP-ER is applied, eet_i becomes strictly greater than t_2.

Let X denote the earliest start time provided by CNP-EF2 for a given subset Ω'. Let us now apply CP-ER to $t_3 = r_{\Omega'}$ and $t_4 = \bar{d}_{\Omega'}$ as well as the processing time constraint $start(A_i) + p_i = end(A_i)$, until we reach quiescence. Since CP-ER is monotonic, a unique fixpoint can be reached in this manner. At this fixpoint, either $d^+(A_i, t_3, t_4) \le 0$ or $eet_i \ge t_4 + d^+(A_i, t_3, t_4)/c_i$.

This implies $d^+(A_i, t_3, t_4) \le \max(0, c_i(eet_i - t_4))$. Then, since $t_4 \le t_2 < eet_i$, this implies $d^+(A_i, t_3, t_4) \le c_i(eet_i - t_4)$.

However, $d^+(A_i, t_3, t_4) \ge e_{\Omega'} + c_i p_i^+(t_3) - C(t_4 - t_3)$, which implies $d^+(A_i, t_3, t_4) > c_i(X - t_4 - 1 + p_i^+(t_3))$. This implies that at the fixpoint we have $(X - t_4 - 1 + p_i^+(t_3)) < eet_i - t_4$. Hence at the fixpoint $X + p_i^+(t_3) \le eet_i$. Since eet_i is strictly greater than t_3, $p_i^+(t_3) = p_i - \max(0, t_3 - r_i)$. Hence at the fixpoint we have $X - \max(0, t_3 - r_i) \le r_i$, which implies either $X \le r_i$ or $X \le t_3$. But $rest(\Omega', c_i) > 0$ implies $X > t_3$. Consequently, when the fixpoint of CP-ER is reached, the obtained earliest start time r_i is at least as strong as X. □

Proposition 72. *CNP-TT* $\le$ *CNP-ER*

Proof. Assume the application of CNP-TT results in an update of the earliest start and end times of A_i. Let r_i and eet_i denote the initial earliest start and end times of A_i. Since CNP-TT updates r_i and eet_i, there exists a time point t in $[r_i, eet_i)$ such that $\sum_{A_j \ne A_i} E(A_j, t)$ exceeds $C - c_i$.

For each A_j with a non-null contribution in this sum, $lst_j \le t < eet_j$. This implies $W_{Sh}(A_j, t, t+1) = c_j$. Hence, $d^+(A_i, t, t+1)$ is equal to $\sum_{A_j \ne A_i} E(A_j, t) + c_i(eet_i - t) - C > 0$. Hence CNP-ER deduces $end(A_i) \ge t + 1 + (eet_i - t) + (\sum_{A_j \ne A_i} E(A_j, t) - C)/c_i$, which strictly exceeds the current value of eet_i. Consequently, when the applications of CNP-ER reach quiescence, the obtained earliest start and end times of A_i are at least as good as those that are given by CNP-TT. □

Remark: In this proof, we have applied CNP-ER to intervals of the form $[t, t+1)$. In fact, since $E(A_j, t)$ is constant between lst_j and eet_j and since $p_i^+(t_1) = eet_i - t_1$ for any t_1 in $[r_i, eet_i)$, one can limit the application of CNP-ER to intervals $[t_1, t_2)$ of the form $[lst_j, eet_k)$ or $[r_i, eet_k)$ with $A_j \neq A_i$ (and no constraint on A_k).

Proposition 73. *CP-TT* $\leq$ *CP-ER*

Proof. Similar to the above, using the fact that an interruptible activity A_j contributes to the Time-Table if and only if $start(A_j)$, $proc(A_j)$ and $end(A_j)$ are bound to values such that the relation $start(A_j) + proc(A_j) = end(A_j)$ holds. □

10.2.5 The One-Machine Non-Preemptive Case

Proposition 74.
FE-TT = *CP-TT* $\leq$ *FE-EF* = *1P-EF* = *CP-ER*
CP-DISJ $\leq$ *FE-EF* = *1P-EF* = *CP-ER*
CNP-EF1 $\leq$ *CP-ER* $\leq$ *CNP-ER*
CNP-EF2 $\leq$ *CP-ER* $\leq$ *CNP-ER*
CNP-TT $\leq$ *CNP-ER*

Proof. See Sections 10.2.3 and 10.2.4. □

Proposition 75. *CNP-TT* $\leq$ *CNP-DISJ*

Proof. See Section 8.1.2 □

Proposition 76.
FE-TT = *CP-TT* $\leq$ *CNP-TT*
CP-DISJ $\leq$ *CNP-DISJ* $\leq$ *CNP-ER-DISJ*
FE-EF = *1P-EF* $\leq$ *1NP-EF* = *CNP-EF1*
CNP-DISJ $\leq$ *1NP-NFNL*
CNP-NFNL $\leq$ *1NP-NFNL*

Proof. Each of these relations is obvious when we assume that the activities have fixed processing times. □

Proposition 77. *FE-EF* = *1P-EF* = *1NP-EF* = *CNP-EF1* = *CP-ER*

Proof. This follows from the previous dominance relations. An alternative proof of 1P-EF = 1NP-EF (under the assumption that activities have fixed processing times) appears in Section 8.2.4. This proof shows that the difference between the two rules serves only to avoid repeated iterations of the algorithm. □

Proposition 78. *CNP-DISJ* $\leq$ *CNP-ER*

Proof. Let's assume that $r_i + p_i + p_j > \bar{d}_j$ and that the adjustment has some impact on r_i for instance, *i.e.*, $r_i < r_j + p_j$. Let us compute the value $W_{Sh}(A_j, t_1, t_2)$ with $t_1 = r_i$ and $t_2 = r_i + p_i$.

$$W_{Sh}(A_j, t_1, t_2) = \min(t_2 - t_1, p_j^+(t_1), p_j^-(t_2))$$
$$p_j^+(t_1) = \max(0, p_j - \max(0, t_1 - r_j))$$
$$p_j^-(t_2) = \max(0, p_j - \max(0, \bar{d}_j - t_2))$$

It is clear that each of the three terms $t_2 - t_1$, $p_j^+(t_1)$ and $p_j^-(t_2)$ is positive. Hence $W_{Sh}(A_j, r_i, r_i + p_i) > 0$. So $d^+(A_i, t_1, t_2) > 0$ and, therefore, $end(A_i) > r_i + p_i$, which in turn leads to an improvement of at least 1 for r_i. Since the same reasoning can be applied while $r_i < r_j + p_j$, we are sure that after some application of the rule, we have $r_i \geq r_j + p_j$ which corresponds to the adjustment computed by CNP-DISJ. □

Proposition 79. *CP-TT $\leq$ CP-DISJ*

Proof. When CP-TT is used (rather than CNP-TT), an activity A_j contributes to the Time-Table if and only if $start(A_j)$, $proc(A_j)$ and $end(A_j)$ are bound to values such that the relation $start(A_j) + proc(A_j) = end(A_j)$ holds. If the earliest start time or end time of an activity A_i is modified by CP-TT, then there exists such an activity A_j and a time t with $r_i \leq t < r_i + p_i$ and $r_j \leq t < \bar{d}_j = r_j + p_j$. Then the second and the fourth disjuncts of the disjunctive constraint

$$\begin{array}{ll} & [start(A_i) + p_i + p_j \leq end(A_i)] \\ \vee & [start(A_i) + p_i + p_j \leq end(A_j)] \\ \vee & [start(A_j) + p_i + p_j \leq end(A_i)] \\ \vee & [start(A_j) + p_i + p_j \leq end(A_j)] \end{array}$$

are discarded and eet_i is updated to the minimum of $r_i + p_i + p_j$ and $r_j + p_i + p_j$. Since A_i is not interruptible, r_i is updated to the minimum of $r_i + p_j$ and $r_j + p_j$. In both cases, this new value is greater than the initial value of r_i. Since CP-TT and CP-DISJ are both monotonic and since whenever CP-TT updates r_i or eet_i, CP-DISJ also does, it can be concluded that CP-TT $\leq$ CP-DISJ. $\square$

10.3 Non-Dominance Relations

Section 10.2 states a number of dominance relations between constraint propagation rules. In this section, we provide examples that show that some rules are not dominated by others. Each example is defined by the list of the cases (FE, CP, 1P, CNP, 1NP) being studied, the resource capacity C, and a set of activities $\{A_i, A_j, \ldots\}$ with given earliest and latest start and end times, processing times, and capacity requirements.

For each example, each rule that performs some deduction is not dominated by the conjunction of those rules that perform no deduction. In particular, each rule that detects a contradiction (*e.g.*, when the domain of some variable becomes empty) is not dominated by the conjunction of the rules that perform no deduction.

10.3.1 General Counterexamples

This section provides a series of counterexamples with one machine and $\forall i, r_i + p_i = eet_i \wedge lst_i + p_i = \bar{d}_i$. These counterexamples are fairly general as they apply in all the considered cases FE, CP, 1P, CNP and 1NP. The non-dominance relations that are established by the first and the second counterexamples are actually of interest in the five cases, while the non-dominance relations established by the following counterexamples are of interest in the non-preemptive cases CNP and 1NP.

Counterexample A for FE, CP, 1P, CNP, 1NP

$$\begin{array}{l} C = 1 \\ p_i = p_j = p_k = p_l = 1 \\ r_i = r_j = r_k = r_l = 0 \\ eet_i = eet_j = eet_k = eet_l = 1 \\ lst_i = 100, lst_j = lst_k = lst_l = 2 \\ \bar{d}_i = 101, \bar{d}_j = \bar{d}_k = \bar{d}_l = 3 \end{array}$$

Time-Table constraints and disjunctive constraints deduce nothing. All the Edge-Finding rules, the CP-ER rule and the CNP-ER rule deduce either that A_i cannot start before 3 or that A_i cannot end before 4. CNP-NFNL, 1NP-NFNL and CNP-ER-DISJ deduce only that A_i cannot start before 1. When r_i and eet_i are set to respectively 1 and 2, Time-Table constraints, disjunctive constraints, CNP-NFNL, 1NP-NFNL and CNP-ER-DISJ make no additional deduction and CNP-EF2 no longer applies. Hence this counterexample shows:

- that Time-Tables and disjunctive constraints do not dominate any other technique;
- that CNP-NFNL, 1NP-NFNL and CNP-ER-DISJ (even with the help of Time-Tables and disjunctive constraints) do not dominate any of FE-EF, 1P-EF, 1NP-EF, CNP-EF1, CNP-EF2, CP-ER and CNP-ER;
- that the non-monotonic CNP-EF2 rule (even with the help of CNP-NFNL, 1NP-NFNL, CNP-ER-DISJ, Time-Tables and disjunctive constraints) does not dominate any of FE-EF, 1P-EF, 1NP-EF, CNP-EF1, CP-ER and CNP-ER.

Counterexample B for FE, CP, 1P, CNP, 1NP

$$\begin{aligned}
&C = 1\\
&p_i = 10, p_j = 1\\
&r_i = r_j = 0\\
&eet_i = 10, eet_j = 1\\
&lst_i = 10, lst_j = 9\\
&\bar{d}_i = 20, \bar{d}_j = 10
\end{aligned}$$

The Time-Table propagation rules FE-TT, CP-TT and CNP-TT deduce nothing. All the other rules deduce either that A_i cannot start before 1 or that A_i cannot end before 11. Hence, the Time-Table propagation rules never dominate any other rule.

Counterexample C for CNP, 1NP

$$\begin{aligned}
&C = 1\\
&p_i = p_j = 5\\
&r_i = 1, r_j = 0\\
&eet_i = 6, eet_j = 5\\
&lst_i = 95, lst_j = 0\\
&\bar{d}_i = 100, \bar{d}_j = 5
\end{aligned}$$

All propagation rules deduce $start(A_i) \geq 5$ or $end(A_i) \geq 10$ except CNP-EF2 which is blocked by the condition $r_i \leq r_\Omega$. This proves that CNP-EF2 dominates no other rule.

Counterexample D for CNP, 1NP

$$\begin{aligned}
&C = 1\\
&p_i = 11, p_j = 10\\
&r_i = 0, r_j = 10\\
&eet_i = 11, eet_j = 20\\
&lst_i = 30, lst_j = 10\\
&\bar{d}_i = 41, \bar{d}_j = 20
\end{aligned}$$

The CNP-NFNL rule deduces nothing while all the other rules deduce either that A_i cannot start before 20 or that A_i cannot end before 21. Let us remark that CNP-NFNL does not apply only because the condition $r_\Omega \leq r_i$ is not satisfied.

Counterexample E for CNP, 1NP

$$\begin{aligned}
&C = 1\\
&p_i = p_j = 2, p_k = 1\\
&r_i = r_j = 0, r_k = 1\\
&eet_i = eet_j = 2,\ eet_k = 2\\
&lst_i = lst_j = 3,\ lst_k = 99\\
&\bar{d}_i = \bar{d}_j = 5, \bar{d}_k = 100
\end{aligned}$$

This example shows that the "Not-First" and "Not-Last" rules are not dominated by the conjunction of all other rules. Indeed, 1NP-NFNL and CNP-NFNL deduce that A_k cannot start before 2 while all the other rules deduce nothing.

Counterexample F for CNP, 1NP

$$\begin{aligned}
&C = 1\\
&p_i = 3, p_j = 2\\
&r_i = 0, r_j = 2\\
&eet_i = 3,\ eet_j = 4\\
&lst_i = 2,\ lst_j = 98\\
&\bar{d}_i = 5, \bar{d}_j = 100
\end{aligned}$$

The rules CNP-TT, CNP-ER, CNP-ER-DISJ, CNP-DISJ, 1NP-NFNL and CNP-NFNL deduce that A_j cannot start before 3 while all other rules deduce nothing.

Counterexample G for CNP, 1NP

$$\begin{aligned}
&C = 1\\
&p_i = 3, p_j = 1, p_k = 3, p_l = 3\\
&r_i = 2, r_j = 2, r_k = 0, r_l = 1\\
&eet_i = 5,\ eet_j = 3,\ eet_k = 3,\ eet_l = 4\\
&lst_i = 97,\ lst_j = 8,\ lst_k = 8,\ lst_l = 6\\
&\bar{d}_i = 100, \bar{d}_j = 9, \bar{d}_k = 11, \bar{d}_l = 9
\end{aligned}$$

The CNP-ER-DISJ rule deduces $start(A_i) \geq 4$. The 1NP-NFNL and CNP-NFNL rules deduce $start(A_i) \geq 3$. The other rules deduce nothing. The reader can verify than when r_i is set to 3, none of the fifteen rules apply. Even CNP-ER-DISJ cannot deduce $start(A_i) \geq 4$ when $r_i = 3$. This illustrates the non-monotonic behavior of CNP-ER-DISJ.

10.3.2 A One-Machine Preemptive Counterexample

Counterexample H for CP, 1P

$$
\begin{aligned}
&C = 1\\
&p_i = p_j = 1, p_k = 6\\
&r_i = 2, r_j = 5, r_k = 0\\
&eet_i = 3,\ eet_j = 6,\ eet_k = 7\\
&lst_i = 2,\ lst_j = 5,\ lst_k = 1\\
&\bar{d}_i = 3, \bar{d}_j = 6, \bar{d}_k = 8
\end{aligned}
$$

This example applies only in the preemptive case since eet_k is not equal to $r_k + p_k$. All the preemptive rules deduce that A_k cannot end before 8, except CP-DISJ, which deduces nothing.

10.3.3 Cumulative Counterexamples

Counterexample I for CP, CNP

$$
\begin{aligned}
&C = 2\\
&c_i = c_j = c_k = 1\\
&p_i = p_j = p_k = 1\\
&r_i = r_j = r_k = 0\\
&eet_i = eet_j = eet_k = 1\\
&lst_i = lst_j = lst_k = 0\\
&\bar{d}_i = \bar{d}_j = \bar{d}_k = 1
\end{aligned}
$$

This example shows that, in the cumulative case, the disjunctive constraints CP-DISJ, CNP-DISJ and CNP-ER-DISJ cannot be used alone. Indeed, all the variables are instantiated, the resource constraint is violated, and these rules do not apply. All the other cumulative constraint propagation rules detect the contradiction.

Remark: The generalization of CNP-DISJ to triples of activities with $c_i + c_j + c_k > C$ (*cf.*, Section 10.1.2) also enables the detection of the contradiction.

Counterexample J for CP, CNP

$$
\begin{aligned}
&C = 5\\
&c_i = c_j = c_k = 2\\
&p_i = p_j = 2, p_k = 8\\
&r_i = r_j = 3, r_k = 0\\
&eet_i = eet_j = 5,\ eet_k = 8\\
&lst_i = lst_j = 3,\ lst_k = 0\\
&\bar{d}_i = \bar{d}_j = 5, \bar{d}_k = 8
\end{aligned}
$$

This example shows that, in the cumulative case, the disjunctive constraints CP-DISJ, CNP-DISJ and CNP-ER-DISJ, the Edge-Finding rules FE-EF and CNP-EF1, and the “Not-First, Not-Last” rules CNP-NFNL cannot be used alone. Indeed, all the variables are instantiated, the resource constraint is violated, and these rules do not apply. All the other cumulative constraint propagation rules, except FE-TT, detect the contradiction.

Remark: The generalization of CNP-DISJ to triples of activities with $c_i + c_j + c_k > C$ (*cf.*, Section 10.1.2) also enables the detection of the contradiction.

Counterexample K for CP, CNP

$$\begin{aligned}
&C = 6\\
&c_i = 2, c_j = 5\\
&p_i = 6, p_j = 4\\
&r_i = 2, r_j = 0\\
&eet_i = 8, eet_j = 4\\
&lst_i = 94, lst_j = 5\\
&\bar{d}_i = 100, \bar{d}_j = 9
\end{aligned}$$

The CP-DISJ and the CNP-DISJ rules deduce either that $start(A_i) \geq 4$ or that $end(A_i) \geq 10$ while all the other rules deduce nothing. This shows that in the cumulative case, the disjunctive rules are not dominated by any of the other rules. This results from the fact that the disjunctive rules apply even when $c_i + c_j$ is just a little greater than C.

Counterexample L for FE, CP, CNP

$$\begin{aligned}
&C = 5\\
&c_i = c_j = c_k = 2\\
&p_i = 22, p_j = p_k = 24\\
&r_i = r_j = r_k = 0\\
&eet_i = 22, eet_j = eet_k = 24\\
&lst_i = 78, lst_j = lst_k = 2\\
&\bar{d}_i = 100, \bar{d}_j = \bar{d}_k = 26
\end{aligned}$$

FE-TT, CP-TT, CP-DISJ, CNP-DISJ, CNP-ER-DISJ deduce nothing. CNP-TT and CNP-NFNL both deduce that A_i cannot start before 24. CNP-EF1 and CNP-EF2 deduce only that A_i cannot start before 9.

FE-EF deduces that A_i cannot end before 28. In the FE and CP cases, this does not relaunch any propagation, even if FE-TT, CP-TT and CP-DISJ are active. In the non-preemptive case, this implies that A_i cannot start before 6. Then FE-EF stops.

CP-ER with $t_1 = 0$ and $t_2 = 26$ provides $end(A_i) \geq 26 + (96 + 44 - 130)/2 = 31$. In the non-preemptive case, this implies $start(A_i) \geq 9$. Then CP-ER enters in a series of iterations:

- CP-ER with $t_1 = 9$ and $t_2 = 26$ provides $end(A_i) \geq 26 + (60 + 44 - 85)/2$, hence $end(A_i) \geq 36$, which in the non-preemptive case implies $start(A_i) \geq 14$.
- Then CP-ER with $t_1 = 14$ and $t_2 = 26$ provides $end(A_i) \geq 26 + (40 + 44 - 60)/2 = 38$, which in the non-preemptive case implies $start(A_i) \geq 16$.
- Then CP-ER with $t_1 = 16$ and $t_2 = 26$ provides $end(A_i) \geq 26 + (32 + 44 - 50)/2 = 39$, which in the non-preemptive case implies $start(A_i) \geq 17$.
- Then CP-ER with $t_1 = 17$ and $t_2 = 26$ provides $end(A_i) \geq 26 + (28 + 44 - 45)/2$, hence $end(A_i) \geq 40$, which in the non-preemptive case implies $start(A_i) \geq 18$.
- Then CP-ER stops with this result. It can be seen that there exists a partially elastic schedule of A_j and A_k that enables A_i to execute between 18 and 40.

Similarly, CNP-ER enters in a series of iterations:

- CNP-ER with $t_1 = 2$ and $t_2 = 24$ provides $end(A_i) \geq 24 + (88 + 40 - 110)/2 = 33$, which implies $start(A_i) \geq 11$.
- Then CNP-ER with $t_1 = 11$ and $t_2 = 24$ provides $end(A_i) \geq 24 + (52 + 44 - 65)/2$, hence $end(A_i) \geq 40$, which implies $start(A_i) \geq 18$.
- Then CNP-ER with $t_1 = 18$ and $t_2 = 24$ provides $end(A_i) \geq 24 + (24 + 44 - 30)/2 = 43$, which implies $start(A_i) \geq 21$.
- Then CNP-ER with $t_1 = 21$ and $t_2 = 24$ provides $end(A_i) \geq 24 + (12 + 44 - 15)/2$, hence $end(A_i) \geq 45$, which implies $start(A_i) \geq 23$.
- Then CNP-ER with $t_1 = 23$ and $t_2 = 24$ provides $end(A_i) \geq 24 + (4 + 44 - 5)/2$, hence $end(A_i) \geq 46$, which implies $start(A_i) \geq 24$.

Hence, in the preemptive case, FE-EF is not dominated by CP-TT + CP-DISJ and CP-ER is not dominated by FE-EF + CP-TT + CP-DISJ.

In the non-preemptive case, only CNP-TT, CNP-NFNL and CNP-ER deduce $start(A_i) \geq 24$. CP-ER deduces $start(A_i) \geq 18$. CNP-EF1 and CNP-EF2 deduce $start(A_i) \geq 9$ and FE-EF deduces $start(A_i) \geq 6$.

Counterexample M for CP

$$\begin{aligned}
&C = 3\\
&c_i = 2, c_j = c_k = 1\\
&p_i = 2, p_j = p_k = 4\\
&r_i = 0, r_j = r_k = 1\\
&eet_i = 3, eet_j = eet_k = 5\\
&lst_i = 3, lst_j = lst_k = 1\\
&\bar{d}_i = 6, \bar{d}_j = \bar{d}_k = 5
\end{aligned}$$

CP-TT deduces that A_i must start at time 0 and end at time 6, while FE-TT, FE-EF, CP-ER and CP-DISJ deduce nothing.

10.4 Summary

This section summarizes the results of the previous sections. Each of the FE, CP, 1P, CNP and 1NP cases is considered in turn.

10.4.1 The Fully Elastic Case

The FE-EF rule strictly dominates the FE-TT rule (*cf.*, section 10.2.1 and counterexamples A, B and L).

Let us remark, however, that the FE-TT rule is not really dominated if the same $E(A_i, t)$ variables appear in several constraints.

10.4.2 The Cumulative Preemptive Case

In the cumulative preemptive case, we have the following results:

- FE-TT $\leq$ FE-EF $\leq$ CP-ER
- FE-TT $\leq$ CP-TT

- CP-ER $\not\leq$ FE-EF + CP-TT + CP-DISJ (counterexample L)
- CP-TT $\not\leq$ CP-ER + CP-DISJ (counterexample M)
- CP-DISJ $\not\leq$ CP-ER + CP-TT (counterexample K)
- FE-EF $\not\leq$ CP-TT + CP-DISJ (counterexample A)
- FE-TT $\not\leq$ CP-DISJ (counterexample H)

10.4.3 The One-Machine Preemptive Case

In the One-Machine preemptive case, we have the following results:

- FE-TT = CP-TT
- FE-EF = CP-ER = 1P-EF
- CP-TT $\leq$ 1P-EF
- CP-DISJ $\leq$ 1P-EF
- 1P-EF $\not\leq$ CP-TT + CP-DISJ (counterexample A)
- CP-DISJ $\not\leq$ CP-TT (counterexample B)
- CP-TT $\not\leq$ CP-DISJ (counterexample H)

10.4.4 The Cumulative Non-Preemptive Case

The following results hold:

- FE-TT $\leq$ CP-TT $\leq$ CNP-TT $\leq$ CNP-ER
- FE-TT $\leq$ FE-EF $\leq$ CNP-EF1 $\leq$ CP-ER $\leq$ CNP-ER
- CNP-EF2 $\leq$ CP-ER $\leq$ CNP-ER
- CP-DISJ $\leq$ CNP-DISJ
- CP-TT $\leq$ CP-ER

Many non-dominance relations have been proven in Section 10.3. In particular no other binary relation, comparing one rule with another, holds.

10.4.5 The One-Machine Non-Preemptive Case

The following results hold:

- FE-TT = CP-TT
- FE-EF = 1NP-EF = 1P-EF = CNP-EF1 = CP-ER
- CP-TT $\leq$ CNP-TT $\leq$ CNP-DISJ $\leq$ 1NP-NFNL
- CP-TT $\leq$ CP-ER $\leq$ CNP-ER
- CP-TT $\leq$ CP-DISJ $\leq$ CP-ER $\leq$ CNP-ER
- CNP-EF2 $\leq$ CP-ER $\leq$ CNP-ER
- CNP-TT $\leq$ CNP-DISJ $\leq$ CNP-ER

- CP-DISJ $\leq$ CNP-DISJ $\leq$ CNP-ER-DISJ
- CNP-NFNL $\leq$ 1NP-NFNL

As in the cumulative non-preemptive case, many non-dominance relations have been proven in Section 10.3. In particular no other binary relation, comparing one rule with another, holds.

The following tables summarize the results. For each couple of rules $\mathcal{R}_1$ and $\mathcal{R}_2$, the entry at row $\mathcal{R}_1$ and column $\mathcal{R}_2$ indicates whether $\mathcal{R}_1$ dominates $\mathcal{R}_2$. If $\mathcal{R}_1$ dominates $\mathcal{R}_2$, the entry is "Yes"; otherwise, a list of counterexamples showing that $\mathcal{R}_2$ is not dominated by $\mathcal{R}_1$ is provided.

$\mathcal{R}_1 \backslash \mathcal{R}_2$	FE-TT	FE-EF
FE-TT	Yes	ABL
FE-EF	Yes	Yes

Table 10.1: Comparison in the FE case. Does $\mathcal{R}_1$ dominate $\mathcal{R}_2$?

$\mathcal{R}_1 \backslash \mathcal{R}_2$	FE-TT	CP-TT	CP-DISJ	FE-EF	CP-ER
FE-TT	Yes	JM	BK	ABL	ABJL
CP-TT	Yes	Yes	BK	ABL	ABL
CP-DISJ	HI	HIJM	Yes	AHIL	AHIJL
FE-EF	Yes	JM	K	Yes	JL
CP-ER	Yes	M	K	Yes	Yes

Table 10.2: Comparison in the CP case. Does $\mathcal{R}_1$ dominate $\mathcal{R}_2$?

$\mathcal{R}_1 \backslash \mathcal{R}_2$	FE-TT	CP-DISJ	FE-EF
FE-TT CP-TT	Yes	B	AB
CP-DISJ	H	Yes	AH
FE-EF 1P-EF CP-ER	Yes	Yes	Yes

Table 10.3: Comparison in the 1P case. Does $\mathcal{R}_1$ dominate $\mathcal{R}_2$?

$\mathcal{R}_1 \backslash \mathcal{R}_2$	*FE-TT*	*CP-TT*	*CNP-TT*	*CP-DISJ*	*CNP-DISJ*	*FE-EF*	*CNP-EF1*	*CNP-EF2*	*CNP-NFNL*	*CP-ER*	*CNP-ER*	*CNP-ER-DISJ*
FE-TT	Yes	J	FJL	BK	BFK	ABL	ABL	ABJL	ABEFGL	ABJL	ABFJL	ABFG
CP-TT	Yes	Yes	FL	BK	BFK	ABL	ABL	ABL	ABEFGL	ABL	ABFL	ABFG
CNP-TT	Yes	Yes	Yes	BK	BK	AB	AB	AB	ABEG	AB	AB	ABG
CP-DISJ	I	IJ	FIJL	Yes	F	AIL	AIL	AIJL	AEFGIL	AIJL	AFIJL	AFG
CNP-DISJ	I	IJ	IJL	Yes	Yes	AIL	AIL	AIJL	AEGIL	AIJL	AIJL	AG
FE-EF	Yes	J	FJL	K	FK	Yes	L	JL	EFGL	JL	FJL	FG
CNP-EF1	Yes	J	FJL	K	FK	Yes	Yes	J	EFGL	JL	FJL	FG
CNP-EF2	C	C	CFL	CK	CF	AC	AC	Yes	CEFGL	ACL	ACFL	CFG
CNP-NFNL	D	DJ	DJ	DK	DK	AD	AD	ADJ	Yes	ADJ	ADJ	DG
CP-ER	Yes	Yes	FL	K	FK	Yes	Yes	Yes	EFGL	Yes	FL	FG
CNP-ER	Yes	Yes	Yes	K	K	Yes	Yes	Yes	EG	Yes	Yes	G
CNP-ER-DISJ	I	IJ	IJL	K	K	AIL	AIL	AIJL	EIL	AIJL	AIJL	Yes

Table 10.4: Comparison in the CNP case. Does $\mathcal{R}_1$ dominate $\mathcal{R}_2$?

$\mathcal{R}_1 \setminus \mathcal{R}_2$	*FE-TT*	*CNP-TT*	*CP-DISJ*	*CNP-DISJ*	*FE-EF*	*CNP-EF1*	*CNP-EF2*	*CNP-NFNL*	*1NP-NFNL*	*CNP-ER*	*CNP-ER-DISJ*
FE-TT CP-TT	Yes	F	B	BF	AB	AB	AB	ABEFG	ABEFG	ABF	ABFG
CNP-TT	Yes	Yes	B	B	AB	AB	AB	ABEG	ABEG	AB	ABG
CP-DISJ	Yes	F	Yes	F	A	A	A	AEFG	AEFG	AF	AFG
CNP-DISJ	Yes	Yes	Yes	Yes	A	A	A	AEG	AEG	A	AG
FE-EF 1P-EF CP-ER	Yes	F	Yes	F	Yes	Yes	Yes	EFG	EFG	F	FG
CNP-EF1 1NP-EF	Yes	F	Yes	F	Yes	Yes	Yes	EFG	EFG	F	FG
CNP-EF2	C	CF	C	CF	AC	AC	Yes	CEFG	CEFG	ACF	CFG
CNP-NFNL	D	D	D	D	AD	AD	AD	Yes	D	AD	DG
1NP-NFNL	Yes	Yes	Yes	Yes	A	A	A	Yes	Yes	A	G
CNP-ER	Yes	Yes	Yes	Yes	Yes	Yes	Yes	EG	EG	Yes	G
CNP-ER-DISJ	Yes	Yes	Yes	Yes	A	A	A	E	E	A	Yes

Table 10.5: Comparison in the 1NP case. Does $\mathcal{R}_1$ dominate $\mathcal{R}_2$?

Chapter 11

Propagation of Objective Functions

In our model, a variable *criterion* represents the value taken by the objective function.

$$criterion = F(end(A_1), \dots, end(A_n)) \tag{11.1}$$

Considering the objective constraint and the resource constraints independently is not a problem when F is a "maximum" such as $C_{\max}$ or $T_{\max}$. Indeed, the upper bound on *criterion* is directly propagated on the completion time of each activity, *i.e.*, deadlines are efficiently tightened. The situation is much more complex for sum functions such as $\sum w_i C_i, \sum w_i T_i$ or $\sum w_i U_i$. For these functions, the constraint (11.1) has to be taken into account at each step of the search tree. An efficient constraint propagation technique must consider the resource constraints and the objective constraint simultaneously. In the following sections, we study three objective functions: $\sum w_i U_i$, $\sum T_i$ and the minimization of setup times.

In all cases, we restrict our study to the situation where each activity requires one machine among a set of m parallel identical machines. So we assume that the resource capacity is $cap(R) = C = m$. For total tardiness, we are even more restrictive since we only consider the situation where $m = 1$.

11.1 Total Weighted Number of Late Activities

As far as the $\sum w_i U_i$ objective function is concerned, our resource constraint is strongly related to the classical problem $P|r_i|\sum w_i U_i$. However, in the classical problem, a late job can be scheduled arbitrary late, while in our model a late activity still has to execute before its deadline.

Many researchers have worked on special cases of $P|r_i|\sum w_i U_i$. We first review the known complexity results under several assumptions, (*e.g.*, one machine *vs.* m machines, preemption *vs.* non-preemption, equal processing times *vs.* arbitrary processing times, equal release dates *vs.* arbitrary release dates, equal weights *vs.* arbitrary weights). There are few results for the problem where the number of machines is unknown and most results apply for $Pm|r_i|\sum w_i U_i$. Without loss of generality, we can suppose in $P|r_i|\sum w_i U_i$ that $r_i + p_i \leq d_i$, for all i, since otherwise J_i cannot be done on time.

11.1.1 Complexity Results

Unit or Equal Processing Times

When $\forall i, p_i = p$, most of the problems become polynomially solvable. We refer to Chapters 2 and 3 for a detailed study of these problems.

Unweighted Case

$1|r_i|\sum U_i$ is NP-hard in the strong sense and is therefore unlikely to be solvable in polynomial time [108]. However, some special cases are solvable in polynomial time. Moore's well-known algorithm [170] solves in $O(n \log n)$ steps the special case where release dates are equal $1||\sum U_i$. Moreover, when release and due dates of jobs are ordered similarly ($r_i < r_j \Rightarrow d_i \leq d_j$), the problem is solvable in a quadratic amount of steps [129]. This result of Kise, Ibaraki and Mine has been extended by Dauzère-Péres and Sevaux [81] to the more general case where $[r_i < r_j] \Rightarrow [d_i \leq d_j] \vee [r_j + p_j + p_i > d_i]$.

Weighted Case

The simple problem $1||\sum w_i U_i$ is already NP-hard. However, it can be solved in pseudo-polynomial time by dynamic programming [145]. The same result also applies for any fixed number of machines $Pm||\sum w_i U_i$ [144].

Preemption

On a single machine, relaxing non-preemption is only of interest if release dates are distinct (otherwise preemptive schedules can be transformed into non-preemptive ones by rescheduling jobs in non-increasing order of their completion times). Lawler [142] has proposed a pseudo-polynomial algorithm for the Preemptive One-Machine Problem with weights $1|r_i, pmtn|\sum w_i U_i$. Time and space bounds of this algorithm are respectively $O(n^3 W^2)$ and $O(n^2 W)$, where W is the sum of the weights of the jobs. The time and space bounds of Lawler's algorithm have been improved in [14] for the unweighted case, to respectively $O(n^4)$ and $O(n^2)$. Allowing preemption on parallel machines does not make the problem easier since Du and others have shown that $P2|pmtn, r_j|\sum U_j$ is NP-hard [92].

Machines	Pmtn.	Release Dates	Proc. Times	Weights	Complexity Status
1	yes/no	no	any	no	$O(n \log n)$ [170]
1	no	yes	any	no	NP-hard [108]
1	yes	yes	any	yes	$O(n^3 W^2)$ [142]
1	yes	yes	any	no	$O(n^5)$ [142], $O(n^4)$ [14]
1	no	yes	$p_i = p$	no	$O(n^3 \log n)$ [53, 109]
1	no	yes	$p_i = p$	yes	$O(n^7)$ Chapter 2
m	no	yes	$p_i = p$	yes	$O(n^{3m+4})$ Chapter 3
1	yes	yes	$p_i = p$	yes	$O(n^{10})$ Chapter 2
m	yes	yes	$p_i = p$	no	open
m	yes	no	$p_i = p$	yes	$O(n^{2^m m!})$ Chapter 3
m	no	no	$p_i = p$	yes	Assignment Pb. *e.g.*, [86]
m	no	yes	$p_i = 1$	yes	Assignment Pb. *e.g.*, [86]
2	yes	yes	any	no	NP-hard [92]
m	yes/no	no	any	yes	pseudo-polynomial [144]

Table 11.1: Complexity results

11.1.2 A Lower Bound of the Number of Late Activities

Obtaining a good lower bound of the weighted number of late activities is the first step of the constraint propagation process. This lower bound is also a lower bound for the domain of the variable *criterion*. Relaxing non-preemption is a well-known technique to obtain good lower bounds. Unfortunately, the preemptive problems remain difficult (*cf.*, Table 11.1) except for $m = 1$ and equal weights. In such a case, the best available algorithm still runs in $O(n^4)$ and it is hardly usable in practice (*cf.*, the computational tests performed in [14]). A "relaxed preemptive lower bound", *i.e.*, a slightly stronger relaxation than the preemptive relaxation, can be used. As shown below, it can be computed in $O(n^2 \log n)$ and is valid for all type of resource constraints (either with one or several machines).

First, recall that the *fully elastic* relaxation (Section 9.1) transforms a cumulative resource constraint into a preemptive One-Machine constraint. **From now on, we study the Preemptive One-Machine Problem with weights**. The lower bounds and the deductions made on this latter resource constraint will be immediately applied on the initial problem (*e.g.*, if there are 3 late activities on the preemptive One-Machine schedule, there are at least 3 late activities on the cumulative problem; if the earliest start time of the activity A_i is adjusted to x on the preemptive problem, the corresponding adjustment will be $\lceil \frac{x}{m} \rceil$).

Let us recall some well-known results on the One-Machine Problem. Its preemptive relaxation is polynomial and has some very interesting properties:

- There exists a feasible preemptive schedule if and only if over any interval $[t_1, t_2)$, the sum of the processing times of the activities in $\{A_i : [t_1 \leq r_i] \wedge [\bar{d}_i \leq t_2]\}$ is lower than or equal to $t_2 - t_1$. It is well known that relevant values for t_1 and t_2 are respectively the release dates and the deadlines [55].

- Jackson's Preemptive Schedule (JPS) is feasible (*i.e.*, each activity ends before its deadline) if and only if there exists a feasible preemptive schedule.

Now assume that JPS is feasible. Proposition 80 highlights the structure of the time intervals $[t_1, t_2)$ that are **full** on JPS, *i.e.*, such that

$$\sum_{A_i : \left\{ \begin{array}{l} t_1 \leq r_i \\ \bar{d}_i \leq t_2 \end{array} \right.} p_i = t_2 - t_1$$

Proposition 80. *If two overlapping intervals are full, their intersection is full.*

Proof. Let $[\tau_1, \tau_2)$ and$[\theta_1, \theta_2)$ be such intervals and let us assume, without any loss of generality that $\theta_1 \leq \tau_1$ (*cf.*, Figure 11.1). If $\tau_2 < \theta_2$, the lemma obviously holds. Assume then that $\tau_2 \geq \theta_2$. Notice that

$$\{A_i : \theta_1 \leq r_i \wedge \bar{d}_i \leq \theta_2\} \subseteq \{A_i : \theta_1 \leq r_i \wedge \bar{d}_i \leq \tau_2\}$$
$$\{A_i : \tau_1 \leq r_i \wedge \bar{d}_i \leq \tau_2\} \subseteq \{A_i : \theta_1 \leq r_i \wedge \bar{d}_i \leq \tau_2\}$$

Hence, the load of the machine over $[\theta_1, \tau_2)$ is

$$\sum_{\left\{ \begin{array}{l} \theta_1 \leq r_i \\ \bar{d}_i \leq \tau_2 \end{array} \right.} p_i \geq \sum_{\left\{ \begin{array}{l} \theta_1 \leq r_i \\ \bar{d}_i \leq \theta_2 \end{array} \right.} p_i + \sum_{\left\{ \begin{array}{l} \tau_1 \leq r_i \\ \bar{d}_i \leq \tau_2 \end{array} \right.} p_i - \sum_{\left\{ \begin{array}{l} \tau_1 \leq r_i \\ \bar{d}_i \leq \theta_2 \end{array} \right.} p_i$$

Since the load of the machine over $[\theta_1, \tau_2)$ is not greater than $\tau_2 - \theta_1$ and since the loads over $[\tau_1, \tau_2)$ and $[\theta_1, \theta_2)$ are respectively $\tau_2 - \tau_1$ and $\theta_2 - \theta_1$,

$$\sum_{\left\{ \begin{array}{l} \tau_1 \leq r_i \\ \bar{d}_i \leq \theta_2 \end{array} \right.} p_i \geq \theta_2 - \theta_1 + \tau_2 - \tau_1 - (\tau_2 - \theta_1) \geq \theta_2 - \tau_1$$

Consequently, the intersection is also full. □

We introduce a decision variable x_i per activity that equals 1 when the activity is on-time and 0 otherwise. Notice that if $\bar{d}_i \leq d_i$, A_i is on-time in any solution, *i.e.*, $x_i = 1$. In such a case we adjust the value of d_i to $\bar{d}_i$ (this has no impact on solutions) so that due dates are always smaller than or

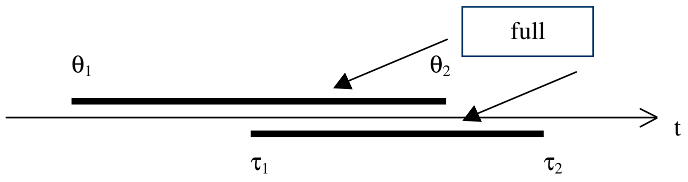


Figure 11.1: Full intervals.

equal to deadlines. We also assume that there is a preemptive schedule that meets all deadlines (if not, the resource constraint does not hold and a backtrack occurs). The following Mixed Integer Program (MIP) computes the minimum weighted number of late activities in the preemptive case:

$$\min \sum_{1}^{n} w_i(1-x_i)$$
$$\text{u.c.} \begin{cases} \forall t_1, \forall t_2 > t_1, \sum_{S(t_1,t_2)} p_i + \sum_{P(t_1,t_2)} p_i x_i \le t_2 - t_1 \\ \forall i \in \{1,\dots,n\}, x_i \in \{0,1\} \end{cases} \tag{11.2}$$

where $S(t_1,t_2)$ is the set of activities that are sure to execute between t_1 and t_2 and where $P(t_1,t_2)$ is the set of activities that are preferred to execute between t_1 and t_2.

$$\begin{aligned} S(t_1,t_2) &= \{A_i : r_i \ge t_1 \wedge \bar{d}_i \le t_2\} \\ P(t_1,t_2) &= \{A_i : r_i \ge t_1 \wedge \bar{d}_i > t_2 \ \wedge d_i \le t_2\} \end{aligned}$$

Actually, it easy to see that the relevant values of t_1 and t_2 correspond respectively to

- the release dates,
- the due dates and the deadlines.

Hence, there are $O(n^2)$ constraints in the MIP. In the following, the notation (t_1,t_2) refers to the resource constraint over the interval $[t_1,t_2)$. We now focus on the **continuous relaxation** of (11.2) in which the following constraints are added : For any activity A_i such that $r_i + p_i > d_i$, *i.e.*, for any late activity, $x_i = 0$.

$$\min \sum_{1}^{n} w_i(1-x_i)$$
$$\text{u.c.} \begin{cases} \forall t_1 \in \{r_i\}, \forall t_2 \in \{\bar{d}_i\} \cup \{d_i\} \ \ (t_2 > t_1) \\ \quad \sum_{S(t_1,t_2)} p_i + \sum_{P(t_1,t_2)} p_i x_i \le t_2 - t_1 \\ \forall i, r_i + p_i > d_i \Rightarrow x_i = 0 \\ \forall i \in \{1,\dots,n\}, x_i \in [0,1] \end{cases} \tag{11.3}$$

(11.3) can be solved in $O(n^2 \log n)$ steps. To achieve this result, we first provide a characterization of a vector that realizes the optimum. From now on, suppose that activities are sorted in non-increasing order of w_i/p_i (this value corresponds to a cost per unit). We introduce the notion of *well-structured* instances. As we will see later on, (11.3) can be solved very efficiently for well-structured instances. Instances that are not well-structured are modified to become well-structured.

Definition 29. *An instance is well-structured if and only if for any pair of activities A_i and A_j with $i < j$,*

$$\begin{cases} d_i < \bar{d}_j \\ d_j < \bar{d}_i \end{cases} \Rightarrow \bar{d}_i \le \bar{d}_j$$

Proposition 81. *For any well-structured instance, the largest vector (according to the lexicographical order) satisfying all the constraints of (11.3) realizes the optimum of (11.3).*

Proof. Let $Y = (Y_1, \ldots, Y_n)$ be the largest vector (according to the lexicographical order) satisfying all the constraints of (11.3), *i.e.*, Y_1 is maximal, Y_2 is maximal (given Y_1), Y_3 is maximal (given Y_1 and Y_2), ..., Y_n is maximal (given $Y_1, \ldots, Y_{n-1}$). Moreover, let $X = (X_1, \ldots, X_n)$ be the largest (according to the lexicographical order) optimal vector of (11.3). Suppose that $X \neq Y$; let then u be the first index such that $X_u < Y_u$. Consider the set K of constraints that are saturated at X.

$$K = \{(t_1, t_2) : A_u \in P(t_1, t_2) \wedge \sum_{S(t_1,t_2)} p_i + \sum_{P(t_1,t_2)} p_i X_i = t_2 - t_1\}$$

If K is empty, then none of the constraints containing the variable x_u is saturated at the point X. $X_u < Y_u$ ensures that $X_u < 1$ and that x_u is not constrained to be equal to 0 due to the second set of constraints; thus, X is not an optimum of (11.3). Hence K is not empty. Let then $(\tau_1, \tau_2) \in K$ be the pair such that τ_1 is maximum and τ_2 is minimum (given τ_1). Let $(\theta_1, \theta_2) \in K$ be the pair such that θ_2 is minimum and θ_1 is maximum (given θ_2).

Suppose that $\theta_1 < \tau_1$. Since $A_u \in P(\tau_1, \tau_2)$ and $A_u \in P(\theta_1, \theta_2)$, we have $\theta_1 < \tau_1 \leq r_u \leq \bar{d}_u \leq \theta_2 < \tau_2$. We claim that the constraint (τ_1, θ_2) is also saturated. To prove our claim, we introduce for each activity A_i two fictive activities A'_i and A'_{n+i}. They have the same earliest start time $r'_i = r'_{n+i} = r_i$, their processing times are respectively $p'_i = p_i X_i$ and $p'_{n+i} = p_i(1 - X_i)$, their latest end times are respectively $\bar{d}'_i = d_i$ and $\bar{d}'_{n+i} = \bar{d}_i$. If A_i is on-time, it corresponds to the fictive activity A'_i and otherwise to A'_{n+i}. The constraints of (11.3) hold, hence there is a preemptive schedule of these activities. On top of that, the time intervals $[\tau_1, \tau_2)$ and $[\theta_1, \theta_2)$ are full. These intervals overlap in time thus, according to Proposition 80, the interval $[\tau_1, \theta_2)$ is full; which leads $(\tau_1, \theta_2) \in K$ for the initial problem. This, together with $\theta_1 < \tau_1$ contradicts our hypothesis on the choice of θ_2.

Now suppose that $\theta_1 = \tau_1 = t_1$ and $\theta_2 = \tau_2 = t_2$. The pair (t_1, t_2) is the unique minimal saturated constraint containing the variable x_u. We claim that among activities in $P(t_1, t_2)$, there is one, say A_v, such that $v > u$ and $X_v > 0$ (otherwise we could prove, because $X_u < Y_u$, that X_u can be increased; which contradicts the fact that X is optimal). Consider now X' the vector defined as follows: $\forall i \notin \{u, v\}, X'_i = X_i$ and $X'_u = X_u + \epsilon/p_u$ and $X'_v = X_v - \epsilon/p_v$, where $\epsilon > 0$ is a small value such that $\epsilon \leq p_u(1 - X_u)$, $\epsilon \leq p_v X_v$ and such that for any non-saturated constraint (t'_1, t'_2),

$$\epsilon + \sum_{S(t'_1,t'_2)} p_i + \sum_{P(t'_1,t'_2)} p_i X_i \leq t'_2 - t'_1$$

First recall that activities are sorted, hence $\epsilon w_u/p_u - \epsilon w_v/p_v \geq 0$. Thus, $\sum w_i(1 - X'_i) \leq \sum w_i(1 - X_i)$. Moreover, X' is "better" for the lexicographical order than X. Second, because of the definition of ϵ, the constraints that were not saturated for X are not violated for X'. Third we claim that the saturated constraints in K (for the vector X) that contain the variables x_u also contain x_v. Indeed, because A_u and A_v both belong to $P(t_1, t_2)$, we have $d_u < \bar{d}_v$ and $d_v < \bar{d}_u$. On top of that, $u < v$ thus, because the instance is well-structured, we have $\bar{d}_u \leq \bar{d}_v$. Hence, a saturated constraint that contains x_u (and hence include the interval $[t_1, t_2)$) also contains x_v. As a consequence X' meets all constraints. This contradicts our hypothesis on X. □

Proposition 81 induces a simple algorithm (Algorithm 17) to compute the optimum X of (11.3). Each time, we compute the maximum resource constraint violation if the activity is fully on-time (lines 6–18). Given this violation, the maximum value X_i that the variable x_i can take is computed. This algorithm runs in $O(n^4)$ since there are n activities A_i and since for each of them $O(n^2)$ violations are computed, each of them in linear time.

Algorithm 18 makes the same computation as Algorithm 17 but uses JPS to compute the violation. The procedure "*ComputeJPS*" of Algorithm 18 is called for several values of i. It computes the JPS of the fictive activities $A'_1, \ldots, A'_n, A'_{n+1}, \ldots, A'_{2n}$ as defined in the proof of Proposition 81. *EndTimeJPS[k]* is the end time of A'_k on JPS. If the processing time of one of these activities is

Algorithm 17 An $O(n^4)$ algorithm to compute the violations

```
1: for $i := 1$ to $n$ do
2:   $X_i := 0.0$
3: end for
4: for $i := 1$ to $n$ do
5:   if $r_i + p_i \leq d_i$ (otherwise $A_i$ is late, i.e., $X_i := 0.0$) then
6:     $X_i := 1.0$, $Violation := 0.0$
7:     for all constraint $(t_1, t_2)$ s.t. $t_1 \in \{r_x\}, t_2 \in \{\bar{d}_x\} \cup \{d_x\}$ do
8:       $total := 0.0$
9:       for $u := 1$ to $n$ do
10:        if $t_1 \leq r_u$ and $\bar{d}_u \leq t_2$ (i.e., $A_i \in S(t_1, t_2)$) then
11:          $total := total + p_u$
12:        end if
13:        if $t_1 \leq r_u$ and $d_u \leq t_2 < \bar{d}_u$ (i.e., $A_i \in P(t_1, t_2)$) then
14:          $total := total + p_u X_u$
15:        end if
16:      end for
17:      $Violation := \max(Violation, total - (t_2 - t_1))$
18:    end for
19:    $X_i := (p_i - Violation)/p_i$
20:  end if
21: end for
```

null then its end time is arbitrarily defined as its release date. JPS can be built in $O(n \log n)$ [54]. Algorithm 18 then runs in $O(n^2 \log n)$.

Proof of the correctness of Algorithm 18. By induction. Suppose that at the beginning of iteration i (line 4), the first coordinates $X_1, ..., X_{i-1}$ are exactly equal to those of Y, the maximal vector (according to the lexicographical order) satisfying the constraints of (11.3). Consider the case $Y_i = 1$. Then, because of the structure of (11.3), there exists a feasible preemptive schedule of the fictive activities where the processing time of A'_i is p_i. Thus, the JPS computed line 7 is also feasible; which means that no violation occurs. Hence, $X_i = 1$ (line 12). Now assume that $Y_i < 1$.

- We first prove that $X_i \leq Y_i$. Since $Y_i < 1$, the violation computed by Algorithm 17 at step i is positive. Let then (t_1, t_2) be the constraint that realizes this violation. We then have

$$Y_i = \frac{1}{p_i}\left(t_2 - t_1 - \sum_{A_u \in S(t_1,t_2)} p_u - \sum_{A_u \in P(t_1,t_2) \wedge u < i} p_u X_u\right)$$

  Consider the subset of the fictive activities $A'_1, ..., A'_n, A'_{n+1}, ..., A'_{2n}$ that have a release date greater than or equal to t_1 and a deadline lower than or equal to t_2. They cannot be completed before t_1 plus the sum of the processing times of these activities, i.e., before

$$\begin{aligned} & t_1 + \sum_{A'_u : \begin{cases} t_1 \leq r_u \\ \bar{d}_u \leq t_2 \end{cases}} p_u \\ = \; & t_1 + \sum_{A_u \in S(t_1,t_2)} p_u + \sum_{\begin{cases} A_u \in P(t_1,t_2) \\ u < i \end{cases}} p_u X_u + p_i \end{aligned}$$

Algorithm 18 An $O(n^2 \log n)$ algorithm to compute the violations

```
for $i := 1$ to $n$ do
    $X_i := 0.0$
end for
for $i := 1$ to $n$ do
    if $r_i + p_i \leq d_i$ then
        $X_i := 1.0$, $\mathit{ViolationJPS} := 0.0$
        $\mathit{ComputeJPS}$
        for all fictive activities $A'_k$ $(1 \leq k \leq 2n)$ do
            $\mathit{ViolationJPS} := \max(\mathit{ViolationJPS}, \mathit{EndTimeJPS}[k] - \bar{d}'_k)$
        end for
        $X_i := (p_i - \mathit{ViolationJPS})/p_i$
    end if
end for
```

Hence, the violation on JPS is at least

$$t_1 + \sum_{A_u \in S(t_1,t_2)} p_u + \sum_{\begin{cases} A_u \in P(t_1,t_2) \\ u < i \end{cases}} p_u X_u + p_i - t_2$$

Hence $X_i \leq Y_i$.

- We now prove that $Y_i \leq X_i$. We have proven that $X_i \leq Y_i < 1$ and thus, *ViolationJPS* is strictly positive. Assume that the maximum violation on JPS is realized by A'_k, $1 \leq k \leq 2n$. The violation is $\mathit{EndTimeJPS}[k] - \bar{d}'_k$. Let t_1 be the largest time point lower than or equal to the end time of this activity such that immediately before t_1, JPS is either idle or executing an activity with a larger deadline than $\bar{d}'_k$. According to the particular structure of JPS, t_1 is an earliest start time. Notice that between t_1 and $\bar{d}'_k$, JPS is never idle and the pieces of activities that are processed are exactly those whose release date are greater than or equal to t_1 and whose deadlines are lower than or equal to $\bar{d}'_k$. Since the values of $X_1, \ldots, X_{i-1}$, resulting from the previous iterations, are such that they cannot alone lead to a violation, this necessarily includes A'_i. As a consequence, $\mathit{EndTimeJPS}[k]$ is equal to:

$$t_1 + \sum_{A_u \in S(t_1,\bar{d}'_k)} p_u + \sum_{\begin{cases} A_u \in P(t_1,\bar{d}'_k) \\ u < i \end{cases}} p_u X_u + p_i$$

□

We have presented an efficient algorithm to compute the optimum of (11.3) for well-structured instances. Figure 11.2 illustrates the use of this algorithm for a well-structured instance with 4 activities.

If an instance is not well-structured, it is easy to relax the deadlines so that it becomes well-structured. To do so, an iteration is performed over all pairs of activities A_i, A_j. If $i < j$, $d_i < \bar{d}_j, d_j < \bar{d}_i$ and $\bar{d}_i > \bar{d}_j$ then $\bar{d}_j$ is momentary set to $\bar{d}_i$. After this modification, the instance is well-structured and we can use Algorithm 18 to obtain a lower bound.

Notice that the algorithm does not work for non-well-structured instances. Consider the counter-example of Table 11.2. The optimum of (11.3) is 3 (A_1 , A_2 and A_3 are respectively scheduled over $[2, 3)$, $[0, 1)$ and $[1, 2)$). Unfortunately, the algorithms both find that $X_1 = 1, X_2 = 0$, and $X_3 = 0$, *i.e.*, A_1, A_2 and A_3 are respectively scheduled over $[0, 1)$, $[1, 2)$ and $[2, 3)$.

Act	*r*	*p*	$\bar{d}$	*d*	*w*	*w/p*
A_1	7	2	12	10	10	5.0
A_2	3	7	12	10	30	4.3
A_3	0	5	14	6	20	4.0
A_4	4	3	18	9	5	1.7

Initial instance. In the following, we focus on the fictive activities.

Act	*r*	*p*	$\bar{d}$
A'_1	7	0	10
A'_2	3	0	10
A'_3	0	0	6
A'_4	4	0	9
A'_5	7	2	12
A'_6	3	7	12
A'_7	0	5	14
A'_8	4	3	18

X_1	0.0
X_2	0.0
X_3	0.0
X_4	0.0

0

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A'_7	A'_7	A'_7	A'_6	A'_6	A'_6	A'_6	A'_6	A'_6	A'_6	A'_5	A'_5	A'_7	A'_7	A'_8	A'_8	A'_8

violation = 0

Step 1: *Is there a schedule where* A_1, A_2, A_3, A_4 *end before their latest end time* ? *Yes*.

Act	*r*	*p*	$\bar{d}$
A'_1	7	2	10
A'_2	3	0	10
A'_3	0	0	6
A'_4	4	0	9
A'_5	7	0	12
A'_6	3	7	12
A'_7	0	5	14
A'_8	4	3	18

X_1	**1.0**
X_2	0.0
X_3	0.0
X_4	0.0

0

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A'_7	A'_7	A'_7	A'_6	A'_6	A'_6	A'_6	A'_1	A'_1	A'_6	A'_6	A'_6	A'_7	A'_7	A'_8	A'_8	A'_8

violation = 0

Step 2: *Try to put* A_1 *on time* (*i.e., the processing time of the fictive activities* A'_1 *and* A'_5 *are set to* 2 *and* 0) *and compute JPS. The maximal violation is* 0 *hence*, X_1 = 1.0.

Act	*r*	*p*	$\bar{d}$
A'_1	**7**	**2**	**10**
A'_2	3	7	10
A'_3	0	0	6
A'_4	4	0	9
A'_5	**7**	**0**	**12**
A'_6	3	0	12
A'_7	0	5	14
A'_8	4	3	18

X_1	**1.0**
X_2	**0.7**
X_3	0.0
X_4	0.0

0

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A'_7	A'_7	A'_7	A'_2	A'_2	A'_2	A'_2	A'_2	A'_2	A'_2	**A'_1**	**A'_1**	A'_7	A'_7	A'_8	A'_8	A'_8

violation = 2

Step 3*: Try to put* A_2 *on time (i.e., the processing time of the fictive activities* A'_2 *and* A'_6 *are set to* 7 *and* 0*) and compute JPS. The maximal violation is* 2 *hence, at most* 5 *units of* A_2 *can be on time, i.e.,* X_2 = 5/7.

Act	*r*	*p*	$\bar{d}$
A'_1	**7**	**2**	**10**
A'_2	**3**	**5**	**10**
A'_3	0	5	6
A'_4	4	0	9
A'_5	**7**	**0**	**12**
A'_6	**3**	**2**	**12**
A'_7	0	0	14
A'_8	4	3	18

X_1	**1.0**
X_2	**0.7**
X_3	**0.6**
X_4	0.0

0

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A'_3	A'_3	A'_3	A'_3	A'_3	A'_2	A'_2	A'_2	A'_2	A'_2	**A'_1**	**A'_1**	**A'_6**	A'_6	A'_8	A'_8	A'_8

violation = 2

Step 4*: Try to put* A_3 *on time (i.e., the processing time of the fictive activities* A'_3 *and* A'_7 *are set to* 5 *and* 0*) and compute JPS. The maximal violation is* 2 *hence, at most* 3 *units of* A_3 *can be on time, i.e.,* X_3 = 3/5.

Act	*r*	*p*	$\bar{d}$
A'_1	**7**	**2**	**10**
A'_2	**3**	**5**	**10**
A'_3	**0**	**3**	**6**
A'_4	4	3	9
A'_5	**7**	**0**	**12**
A'_6	**3**	**2**	**12**
A'_7	**0**	**2**	**14**
A'_8	4	0	18

X_1	**1.0**
X_2	**0.7**
X_3	**0.6**
X_4	**0.0**

0

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
A'_3	A'_3	A'_3	A'_2	A'_4	A'_4	A'_4	A'_2	A'_2	A'_2	**A'_2**	**A'_1**	**A'_1**	**A'_6**	**A'_6**	**A'_7**	**A'_7**

violation = 3

Step 5*: Try to put* A_4 *on time (i.e., the processing time of the fictive activities* A'_4 *and* A'_8 *are set to* 3 *and* 0*) and compute JPS. The maximal violation is* 3 *hence,* A_4 *is late, i.e.,* X_4 = 0.

Figure 11.2: Illustration of Algorithm 18

	r_i	$\bar{d}_i$	d_i	p_i	w_i
A_1	0	10	1	1	3
A_2	0	2	1	1	2
A_3	0	3	2	1	2

Table 11.2: A counter-example for non-well-structured instances

11.1.3 Constraint Propagation

In this section, some deduction rules are presented. They determine that some activities can, must or cannot end before their due date. Consider an activity A_u such that $eet_u \leq d_u < \bar{d}_u$; it can be either late or on-time. Our objective is to compute efficiently a lower bound of the weighted number of late activities if A_u is on-time (conversely if A_u is late). If this lower bound is greater than the maximal value in the domain of *criterion* then A_u must be late (conversely on-time). Algorithm 18 could be used to compute such a lower bound. However, it would be called n times, leading to a high overall complexity of $O(n^3 \log n)$. We propose to use a slightly weaker lower bound that can be computed, after some preprocessing, in linear time, for each activity. We will see that the overall domain reduction scheme runs in $O(n^2)$.

First, *deadlines are relaxed* to a very large value (except for the activities that have to be on-time), *i.e.*, late activities can be scheduled arbitrarily late. Once this is done, we compute the optimum X of the linear program obtained with the relaxed deadlines (the instance is well-structured and so we can use Algorithm 18). It is easy to see that this program can be rewritten as follows:

$$\min \sum_1^n w_i(1-x_i)$$
$$\text{u.c.} \begin{cases} \forall r_j, \forall d_k > r_j, & \sum_{A_i \in P(r_j,d_k)} p_i x_i \leq d_k - r_j \\ \forall i, r_i + p_i > d_i \Rightarrow x_i = 0 \\ \forall i, \bar{d}_i \leq d_i \Rightarrow x_i = 1 \\ \forall i \in \{1,\dots,n\}, x_i \in [0,1] \end{cases} \tag{11.4}$$

The second and the third constraint simply take into account the fact that some activities are known to be late (or on-time). Of course, the lower bound provided by (11.4) is weaker than the one provided by (11.3) (because deadlines have been relaxed). However, the computation of X is the basis of our domain reduction scheme.

Late Activity Detection

Let (11.5) be the linear program (11.4) to which the constraint $x_u = 1$ has been added.

$$\min \sum_1^n w_i(1-x_i)$$
$$\text{u.c.} \begin{cases} \forall r_j, \forall d_k > r_j, & \sum_{A_i \in P(r_j,d_k)} p_i x_i \leq d_k - r_j \\ \forall i, r_i + p_i > d_i \Rightarrow x_i = 0 \\ \forall i, \bar{d}_i \leq d_i \Rightarrow x_i = 1 \\ \forall i \in \{1,\dots,n\}, x_i \in [0,1] \\ x_u = 1 \end{cases} \tag{11.5}$$

Assume that there is a feasible solution of (11.5) and let Xo be the optimal vector of (11.5) obtained by Algorithm 17. Propositions 82 and 83 exhibit two relations that X and Xo satisfy. These relations are used to compute a lower bound of the weighted number of late activities.

Proposition 82. $\sum p_i Xo_i \leq \sum p_i X_i$

Proof. Let $G(I, T, E)$ be a bipartite graph, where $I = \{1, ..., n\}$ is a set of vertices corresponding to the activities, where $T = \{t : \min_i r_i \leq t < \max_i d_i\}$ is a set of vertices corresponding to all the "relevant" time-intervals $[t, t+1)$ and where an edge (i, t) belongs to E if and only if A_i can execute on time in $[t, t+1)$ (*i.e.*, $r_i \leq t < d_i$). Consider the network flow (*cf.*, Figure 11.3) built from G by adding:

- two vertices S, P and an edge (P, S),
- for each node i an edge (S, i) whose capacity is (i) upper bounded by p_i and (ii) lower bounded by p_i if $\bar{d}_i \leq d_i$ and by 0 otherwise,
- for each node t an edge (t, P) whose capacity is upper bounded by 1.

A flow f on G corresponds to a preemptive schedule of the activities where $f(S, i)$ units of each activity are on-time and where the remaining units are late; $f(x, y)$ being the value of the flow through the edge (x, y). Hence, to any feasible flow f we can associate the vector X that meets all the constraints of (11.4):

$$X_i = \frac{f(S, i)}{p_i}$$

Conversely, a feasible flow can be derived from any vector X meeting all the constraints of (11.4) such that $p_i X_i$ is integer: Compute the JPS of the activities with a processing time of $p_i X_i$ and set $f(i, t) = 1$ if and only if one unit of J_i executes at time t on JPS.

Suppose that $\sum p_i Xo_i > \sum p_i X_i$ then the flow corresponding to X is not maximal in G and thus there is an augmenting path from S to P. Let then X^+ be the vector corresponding to the augmented flow. Because of the structure of G, $\forall i, X_i^+ \geq X_i$. On top of that there exists l such that $X_l^+ > X_l$. This contradicts proposition 81. □

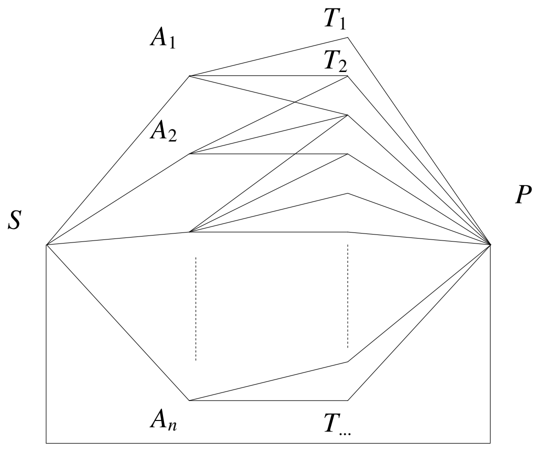


Figure 11.3: Network flow associated to the graph G

Proposition 83. $\forall i \neq u, Xo_i \leq X_i$

Proof. Suppose the proposition does not hold. Let i be the first index such that $i \neq u$ and $Xo_i > X_i$. We modify the instance of the problem by removing the activities A_v with $v > i$ that do not have to be on-time (*i.e.*, $v \neq u$ and $\bar{d}_v > d_v$). The activities that have been removed do not influence Algorithm 17 when computing the i first coordinates of X and of Xo (*i.e.*, for the modified instance, the i first coordinates of the optimum vector are exactly those of X). Now, consider the modified instance. We still have $i \neq u$ and $Xo_i > X_i$. Moreover, the activities that have a greater index than i are on-time. Consider the modified network (Figure 11.4) built from the bipartite graph G by adding:

- three vertices S, S', P and two edges, (S, S') and (P, S),
- for each node v $(v \neq i)$ an edge (S', v) whose capacity is (i) upper bounded by 0 if A_v has to be late and by p_v otherwise and (ii) lower bounded by p_v if A_v has to be on-time and by 0 otherwise,
- an edge (S, i) whose capacity is upper bounded by p_i and lower bounded by 0,
- for each node t an edge (t, P) whose capacity is upper bounded by 1.

For any feasible flow, a vector satisfying all constraints of (11.4) can be built. Conversely, for any vector satisfying all constraints of (11.4), a feasible flow can be built. The flow corresponding to Xo is obviously feasible. Moreover the flow on (P, S) for vector X is greater than or equal to the one for Xo (see Proposition 82). Moreover, the flow on (S, i) for Xo is greater than the one for X. Hence, because of the conservation law at S, the flow that goes over (S, S') is not maximal for Xo. As a consequence, there is an augmenting path from S' to S for the flow corresponding to Xo. Let then Xo^+ be the vector corresponding to the augmented flow. Because of the structure of G, $\forall v \neq i, Xo^+_v \geq Xo_v$. Hence, for any activity that has to be on-time, $Xo^+_v = 1$ and then Xo^+ satisfies all the constraints of (11.5). Moreover it is better than Xo because:

- If the edge (P, S) is in the augmenting path then $\sum p_i Xo^+_i > \sum p_i Xo_i$.
- If the edge (i, S) is in the augmenting path then we claim that Xo^+ is greater, for the lexicographical order, than Xo. Indeed, there is an edge from S' to an activity, say v, in the augmenting path. Hence, A_v does not have to be on-time (otherwise, the edge would be saturated for Xo and it could not belong to the augmenting path). Consequently, $v < i$ and then $Xo^+_v > Xo_v$.

This contradicts the fact that Xo is optimal. □

Thanks to Propositions 82 and 83, we can add constraints $\sum p_i Xo_i \leq \sum p_i X_i$ and $\forall i \neq u, x_i \leq X_i$ to the linear program (11.5). Since we are interested in a lower bound of (11.5), we can also relax the resource constraints. As a consequence, we seek to solve the following program (11.6) that is solved in linear time by Algorithm 19.

$$\min \sum_{1}^{n} w_i(1 - x_i)$$
$$\text{u.c.} \begin{cases} \sum p_i x_i \leq \sum p_i X_i \\ \forall i \neq u, x_i \leq X_i \\ x_u = 1 \\ \forall i \in \{1, \dots, n\}, x_i \in [0, 1] \end{cases} \tag{11.6}$$

Algorithm 19 A linear time algorithm for late activity detection

```
1: for $i := 1$ to $n$ do
2:     $Xo_i := 0.0$
3: end for
4: $Xo_u := 1$
5: $MaxVal := \sum p_i X_i - p_u$
6: for $i := 1$ to $n$ and $i \neq u$ do
7:     $Xo_i := \min(X_i, MaxVal/p_i)$
8:     $MaxVal = MaxVal - p_i * Xo_i$
9: end for
```

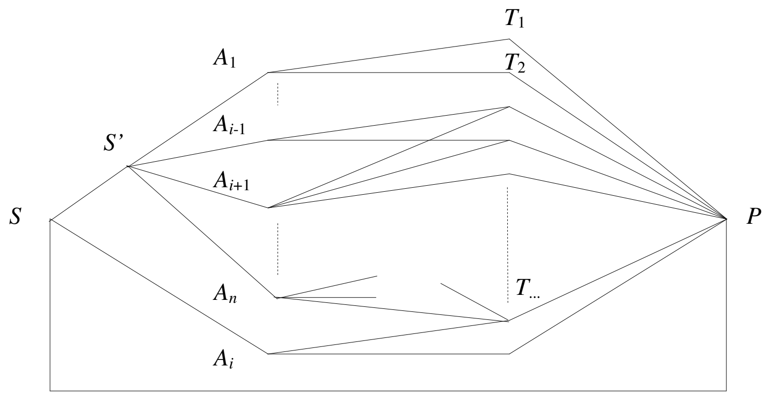


Figure 11.4: Modified network flow associated to the graph G

On-Time Activity Detection

Let A_u be an activity such that $eet_u \leq d_u < \bar{d}_u$. A_u can be either late or on-time. We want to compute a lower bound of the number of late activities if it is late. Let Xl be the optimal vector of the linear program (11.4) to which the constraint $x_u = 0$ has been added. We claim that $\sum p_i Xl_i \leq \sum p_i X_i$ and that $\forall i \neq u, Xl_i \geq X_i$ (proofs are similar to the proofs of Propositions 82 and 83). The same mechanism than for the late activity detection then applies: The new constraints are entered in the linear program while the resource constraints are removed. The resulting linear program can be also solved in linear time.

Example

Consider the instance of Figure 11.2. Deadlines are relaxed. The optimal vector of (11.4) is $X_1 = 1, X_2 = 5/7, X_3 = 3/5, X_4 = 0$. The lower bound is then $(0 * 10)/2 + (2 * 30)/7 + (2 * 20)/5 + (3 * 5)/3 = 21.57$ Assume that the domain of *criterion* is $[22, 26]$. Let us try to put A_4 on-time. The optimum of (11.6) is $Xo_1 = 1, Xo_2 = 5/7, Xo_3 = 0, Xo_4 = 1$. Thus, the new lower bound is $(0 * 10)/2 + (2 * 30)/7 + (5 * 20)/5 + (0 * 5)/3 = 28.57$. Given the domain of *criterion*, we deduce that A_4 must be late.

11.2 Total Tardiness

We propose a set of techniques[1], to propagate the $\sum T_i$ criterion. We restrict our study to the single machine case and unfortunately, extensions to cumulative problems are not straightforward. We also assume that there are no initial deadlines (*i.e.*, those that appear in the following algorihms have been deduced by some domiance rules).

The single machine total tardiness scheduling problem has been widely studied in the literature (see Section 14.2 for a brief review). Many dominance properties have been proposed in the case where $\forall i, r_i = 0$. In comparison, less efficient lower-bounds have been introduced. In this section, we focus on constraint propagation, a purely deductive process in which dominance rules cannot a priori be incorporated. However, we will see that some new dominance rules can be used as a preprocessing step to compute a very efficient lower bound.

11.2.1 Lower Bounds

Relaxing non-preemption is a standard technique to obtain lower bounds for non-preemptive scheduling problems. Unfortunately, the total tardiness problem with no release dates (for which the preemption is not useful) is already NP-Hard, so some additional constraints have to be relaxed to

[1]All the results presented in this section come from [16]

obtain a lower bound in polynomial time. From now on, assume that activities are sorted in **non-decreasing order of due dates**. We first recall Chu's lower bound. Latter we will show how some new dominance properties can be used to improve this lower bound.

Chu [75] has introduced an $O(n \log n)$ lower bound, $\boldsymbol{lb_{Chu}}$, that can be computed as follows. Preemption is relaxed and activities are scheduled according to the SRPT (Shortest Remaining Processing Time) rule. Each time an activity becomes available or is completed, an activity with the shortest remaining processing time among the available and uncompleted activities is scheduled (see Table 11.3 and Figure 11.5). The computation of the lower bound is based on Proposition 84 ([75]).

A_i	r_i	p_i	d_i
A_1	0	5	5
A_2	1	4	6
A_3	3	1	8

Table 11.3: An Instance of $1|r_i|\sum T_i$.

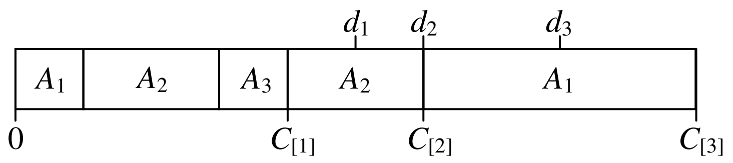


Figure 11.5: An SRPT Schedule of the Instance of Table 11.3.

Proposition 84. *Let $[i]$ denote the index of the activity which is completed in the i^{th} position in the SRPT schedule. The total tardiness is at least*

$$\sum_{i=1}^{n} \max(C_{[i]} - d_i, 0).$$

11.2.2 Improving Lower Bounds With Dominance Properties

A dominance rule is a constraint that can be added to the initial problem without changing the value of the optimum, *i.e.*, there is at least one optimal solution of the problem for which the dominance holds. Dominance rules can be of prime interest since they can be used to reduce the search space. However they have to be used with care since the optimum can be missed if conflicting dominance rules are combined.

Emmons Rules

We recall Emmons Rules and generalize them to take into account release dates. Unfortunately, these rules are valid if preemption is allowed (or, for some of them, if activities have identical processing times). Nevertheless, we will see in Section 11.2.2 that these rules can be used in a pre-processing phase before computing a preemptive lower-bound of the problem.

Initial Emmons Rules

Emmons [95] has proposed a set of dominance rules for the special case where release dates are equal ($1||\sum T_i$). These rules allow us to deduce some precedence relations between activities. Following Emmons notation, $\mathcal{A}_i$ and $\mathcal{B}_i$ are the sets of activities that have to be scheduled, according to the dominance rules, respectively after and before A_i. In the following, we say that A_i precedes A_k when there is an optimal schedule in which A_i precedes A_k and for which all previously mentioned dominance properties hold.

Emmons Rule 1. *$\forall i, k (i \neq k)$, if $p_i \leq p_k$ and $d_i \leq \max(\sum_{A_j \in \mathcal{B}_k} p_j + p_k, d_k)$ then A_i precedes A_k ($A_i \in \mathcal{B}_k$ and $A_k \in \mathcal{A}_i$).*

Emmons Rule 2. *$\forall i, k (i \neq k)$, if $p_i \leq p_k$ and $d_i > \max(\sum_{A_j \in \mathcal{B}_k} p_j + p_k, d_k)$ and $d_i + p_i \geq \sum_{A_j \notin \mathcal{A}_k} p_j$, then A_k precedes A_i.*

Emmons Rule 3. *$\forall i, k (i \neq k)$, if $p_i \leq p_k$ and $d_k \geq \sum_{A_j \notin \mathcal{A}_i} p_j$, then A_i precedes A_k.*

Generalized Emmons Rules ($1|r_i,\ pmtn, \sum T_i$)

Our aim is to generalize Emmons rules to the situation where we have arbitrary release dates. Such a generalization is relatively easy to do if we relax the non-preemption constraint. We will see that the resulting rules can be used to tighten the lower bound of the non preemptive problem (Section 11.2.2).

In the preemptive case, A_i is said to precede A_k if and only if A_k starts after the end of A_i. As for the initial Emmons rules, $\mathcal{A}_i$ and $\mathcal{B}_i$ respectively denote the set of activities that are known to execute after and before A_i.

Note that active schedules are dominant, *i.e.*, we only consider schedules in which activities or pieces of activities cannot be scheduled earlier without delaying another activity. It is easy to see that all active schedules have exactly the same completion time C_{max}. To compute this value, we can build the schedule where activities are scheduled in non decreasing order of release dates. Now we can tighten the deadlines since A_i since it cannot be completed after $C_{max} - \sum_{A_j \in \mathcal{A}_i} p_j$. These deadlines are only valid in the preemtive case.

In the following, we note $C_{max}(E)$ the completion time of active schedules of a subset E of activities (other activities are not considered). As mentioned previously, all active schedules have the same completion time and it can be computed in polynomial time. $C_{max}(E)$ is a lower bound of the maximal completion time of the activities of E in any schedule of $\{A_1, \ldots, A_n\}$. If $\mathcal{B}_i$ is a set of activities that have to be processed before A_i then A_i cannot start before $C_{max}(\mathcal{B}_i)$, *i.e.*, the **release date** r_i can be **adjusted** to $\max(r_i, C_{max}(\mathcal{B}_i))$.

First, we make the following **remark** which allows us to compare the values of the tardiness of an activity A_i in two schedules S and S': Let C_i and C'_i be the completion times of A_i on two preemptive schedules S and S' and let T_i and T'_i be respectively the tardiness of A_i in S and S'.

- If $C'_i < C_i$ and $d_i \leq C_i$ then $T'_i - T_i = \max(C'_i, d_i) - C_i \leq 0$.
- If $d_i \geq \max(C_i, C'_i)$ then A_i is on time in S and in S' and $T'_i - T_i = 0$.

Generalized Emmons Rule 1. *Let A_i and A_k be two activities such that $r_i \leq r_k$, $p_i \leq p_k$ and $d_i \leq \max(r_k + p_k, d_k)$, then A_i precedes A_k ($A_i \in \mathcal{B}_k$ and $A_k \in \mathcal{A}_i$).*

Proof. Consider a schedule S in which activity A_i and activity A_k satisfy the assumptions. First,

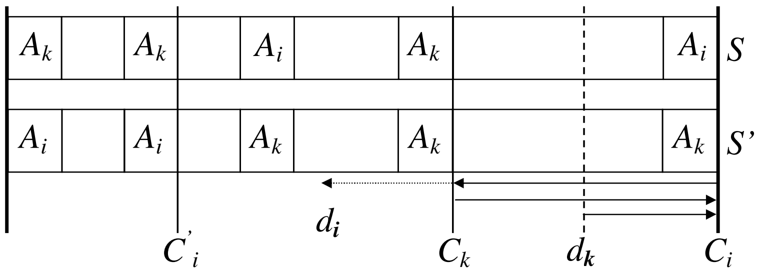


Figure 11.6: Generalized Emmons Rule 1.

assume that activity A_k is completed before activity A_i ($C_k < C_i$). Let us reschedule the pieces of A_i and A_k such that all pieces of A_i are completed before the pieces of A_k (see Figure 11.6). Note that the exchange is valid since r_i is lower than or equal to r_k. We show that this exchange does

not increase total tardiness. Let S' be this new schedule and let C'_i be the completion time of A_i in S'. In S', A_k is completed at time C_i. If $C_i \leq d_i$, the activities A_i and A_k are on time in the two schedules S and S' and the exchange has no effect. From now, we suppose that $d_i < C_i$.
Since the completion times of all other activities remain the same, the difference between the total tardiness of S' and of S is exactly $\Delta = T'_i - T_i + T'_k - T_k$. Following our initial remark, $T'_i - T_i = \max(C'_i, d_i) - C_i$ and $T'_k - T_k = C'_k - \max(C_k, d_k) = C_i - \max(C_k, d_k)$. So, $\Delta = \max(C'_i, d_i) - \max(C_k, d_k)$. Note that all pieces of A_i are completed before C_k in S' and since $p_i \leq p_k$, we have $C'_i \leq C_k$ and $\Delta \leq \max(C_k, d_i) - \max(C_k, d_k)$.
Now consider two cases:

- If $\max(r_k + p_k, d_k) = d_k$ then $d_i \leq d_k$. Together with $C'_i \leq C_k$, this leads to $\Delta \leq 0$.
- If $\max(r_k + p_k, d_k) = r_k + p_k$ then $d_i \leq r_k + p_k \leq C_k$ and $d_k \leq r_k + p_k \leq C_k$. This leads to $\Delta \leq C_k - C_k = 0$.

Now assume, that activity A_i is completed before activity A_k. Rescheduling the pieces of the two activities, such that all pieces of A_i are completed before the pieces of A_k, can only decrease the tardiness of A_i, and leave unchanged the tardiness of A_k.
In all cases, the exchange does not increase total tardiness. □

As shown in Figure 11.7, the Generalized Emmons Rule 1, does not hold in the non-preemptive case. Indeed, A_1 would have to be completed before A_2, which is not true in the non-preemptive case.

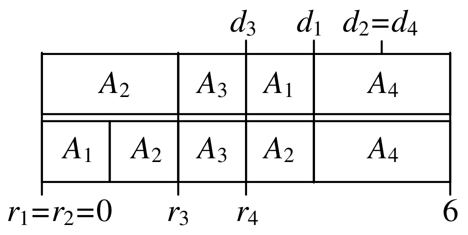


Figure 11.7: An Optimal Non-Preemptive Schedule and an Optimal Preemptive Schedule.

The proof for the two following rules follow the same scheme as above (see [16]).

Generalized Emmons Rule 2. *Let A_i and A_k be two activities such that $r_i \leq r_k$, $d_i \leq d_k$ and $\bar{d}_i \leq d_k + p_k$, then A_i precedes A_k*

Generalized Emmons Rule 3. *Let A_i and A_k be two activities such that $r_i \leq r_k$ and $\bar{d}_i \leq d_k$, then A_i precedes A_k.*

Combining Generalized Emmons Rules

As mentioned previously, dominance properties cannot always be combined. Fortunately, in our case, the Generalized Emmons Rules are compatible, which was already the case for $1||\sum T_i$ as noticed by Emmons [95].

We can use these dominance rules one after the other to adjust the data of an instance: if a set $\mathcal{B}_i$ of activities is proved to precede A_i according to the Generalized Emmons Rules, it is possible to adjust r_i to $r_i = \max(r_i, C_{max}(\mathcal{B}_i))$. By using iteratively these rules, we are sure, that at the end of each iteration, the adjusted instance has the same optimal total tardiness than the original instance, and that we have not introduce any inconsistencies.

We consider activities one after the other and each time we perform adjustments. Now, we can prove that we do not introduce any inconsistencies. Indeed, suppose that at the current iteration, we have for each activity A_i, a set $\mathcal{B}_i$ of activities which are proven to precede A_i, and a set $\mathcal{A}_i$ of activities which are proven to follow A_i. Suppose too that there is no contradiction at this time and that all adjustments are made considering these sets. It means that we have $r_j < r_i$, for any $A_j \in \mathcal{B}_i$ and $r_i < r_j$, for any $A_j \in \mathcal{A}_i$. Suppose now, we want to refine the adjustments for an activity A_k. Let $A_i \notin \mathcal{B}_k$ be an activity which has not yet been proved to precede A_k, but for which, one of the Generalized Emmons Rule has proven that A_i precedes A_k. This implies that $r_i \leq r_k$. According to this rule, we can introduce a contradiction only if there exists an activity A_m such that: A_k precedes A_m and A_m precedes A_i. Because we have made all adjustments implied by the precedence relations already established, we have: A_k precedes A_m implies $r_k < r_m$, and A_m precedes A_i implies $r_m < r_i$, then $r_k < r_m < r_i$, which contradicts the initial assumption $r_i \leq r_k$. Consequently, we will not introduce any inconsistency according to the precedence relations already established.

Each rule can be implemented in $O(n^2)$. The activities are sorted in non-decreasing order of release dates in a heap structure, which runs in $O(n \log n)$. For each activity A_i, we consider the set of activities which are completed before A_i according to the Generalized Emmons Rules. Computing the earliest completion time of this set and adjusting r_i runs in $O(n)$, since the activities are already sorted in non-decreasing order of release dates. The heap structure is maintained in $0(\log n)$. Therefore, we have an overall time complexity of $O(n)$ for each activity. New precedence dominances can be found and the rules might have to be applied again. Each activity can be adjusted at most $n-1$ times. Hence, in the worst case the whole propagation is in $O(n^3)$.

A New Lower Bound

We introduce two propositions, that are valid in the preemptive case only. Proposition 85 is a weak version of the Generalized Emmons Rule 1. Proposition 86 shows that under some conditions, due dates can be exchanged without increasing the objective function.

Proposition 85. *Let A_i and A_k be two activities such that $r_i \leq r_k$, $p_i \leq p_k$ and $d_i \leq d_k$, then there exists an optimal schedule in which A_k starts after the end of A_i.*

Proof. See proof of Generalized Emmons Rule 1. □

Proposition 86. *Let A_i and A_k be two activities such that $r_i \leq r_k$, $p_i \leq p_k$ and $d_i > d_k$. Exchanging d_i and d_k does not increase the optimal total tardiness.*

Proof. Consider an optimal schedule S of the original instance and let C_i and C_k be the completion times of A_i and A_k in S. Let Δ be the difference between the tardiness of S before and after the exchange.

$$\Delta = \max(0, C_i - d_k) + \max(0, C_k - d_i) - \max(0, C_i - d_i) - \max(0, C_k - d_k).$$

First, assume that $C_i < C_k$ and consider the following cases.
If $d_k < d_i \leq C_i$ then

$$\Delta = (C_i - d_k) + (C_k - d_i) - (C_i - d_i) - (C_k - d_k) = 0.$$

If $d_k \leq C_i \leq d_i \leq C_k$ then

$$\Delta = (C_i - d_k) + (C_k - d_i) - 0 - (C_k - d_k) = C_i - d_i \leq 0.$$

If $C_i \leq d_k < d_i \leq C_k$ then $\Delta = 0 + (C_k - d_i) - 0 - (C_k - d_k) = d_i - d_k \leq 0$.
If $C_i \leq d_k \leq C_k \leq d_i$ then $\Delta = 0 + 0 - 0 - (C_k - d_k) \leq 0$.
If $C_i < C_k \leq d_k < d_i$ then $\Delta = 0$.
If $d_k \leq C_i < C_k \leq d_i$ then

$$\Delta = (C_i - d_k) + 0 - 0 - (C_k - d_k) = (C_i - C_k) + (d_k - d_i) < 0.$$

Now assume that $C_k < C_i$. After the exchange of due dates, we exchange the pieces of the activities, such that all pieces of A_i are scheduled before those of A_k. This exchange is valid since $r_i \leq r_k$. In this new schedule, A_i is completed before or at C_k because $p_i \leq p_k$, thus the tardiness of A_i is lower than or equal to $\max(0, C_k - d_k)$. Moreover, A_k is completed at C_i and its tardiness is equal to $\max(0, C_i - d_i)$. Thus $\Delta \leq \max(0, C_i - d_i) + \max(0, C_k - d_k) - \max(0, C_i - d_i) - \max(0, C_k - d_k) \leq 0$. □

These propositions allow us to compute a lower bound lb_1 thanks to the following algorithm.

At each time t, we consider $D = \{A_j / r_j \leq t \wedge p'_j > 0\}$ the set of activities available but not completed at t (p'_j denotes the remaining processing time of A_j at time t). Let A_u be the activity with the shortest remaining processing time, and let A_v be the activity with the smallest due date. If $d_u = d_v$, *i.e.*, A_u has the smallest due date, according to Proposition 85, it is optimal to schedule one unit of this activity. If it is not the case, according to Proposition 86, we exchange its due date with the one of A_v, the activity with the smallest due date. This new instance has an optimal total tardiness lower than or equal to the optimal tardiness of the original problem. In this new problem, A_u has now the smallest due date and the smallest remaining processing time, then it is optimal to schedule one unit of this activity according to Proposition 85. We increase t and we iterate until all activities are completed.

At each time t, we build a new problem by exchanging two due dates. This new problem has an optimal total tardiness lower than or equal to the problem before. Hence, at the end of the algorithm, we obtain an optimal schedule of a problem which has a total tardiness lower than or equal to the optimal tardiness of the original problem. Therefore, it is a lower bound of the original problem. Actually, the only relevant time points are when an activity becomes available or when an activity is completed. And so there are at most $2n$ times t to consider.

We maintain two heaps $heap_p$ and $heap_d$ which contain respectively, the uncompleted activities which are available at time t sorted in non-decreasing order of remaining processing times, and these activities sorted in non-decreasing order of due dates. The insertion, the extraction and the modification of an activity of one heap costs $O(\log n)$. Getting the minimum element of one heap costs $O(1)$. We have at most n insertions (each time an activity becomes available), and at most n extractions (each time an activity is completed). At each time t, we have at most two modifications in $heap_p$ and in $heap_d$ when we must exchange the due dates of two activities. Hence, our algorithm runs in $O(n \log n)$.

Comparison with Chu's Lower Bound

The schedule built by our algorithm is the same as the one built by Chu. The algorithms differ from the assignment of the due dates to the activities. Indeed, in the algorithm of Chu, the assignment of the due dates is performed at a "global" level, whereas in our algorithm, the assignment is performed at a "local" level. We show that lb_1 is strictly better than lb_{Chu}.

Proposition 87. *The lower bound lb_1 obtained by our algorithm is stronger than Chu's lower bound lb_{Chu}.*

Proof. The sets of completion times in the schedules built by our algorithm and by Chu's are the same. The difference lies in the assignment of due dates to the activities. Let $(d_1, \ldots, d_n)$ be the sequence of due dates sorted in non-decreasing order. This is the sequence obtained by the algorithm of Chu. Let $(d_{\sigma(1)}, \ldots, d_{\sigma(n)})$ be the sequence of due dates obtained by our algorithm. Let j be the first index such that $j \neq \sigma(j)$ and let t be the starting time of the piece of activity which is completed at $C_{[j]}$. This piece is a piece of the activity with the shortest remaining processing time at t. We have $d_j \in (d_{\sigma(j+1)}, \ldots, d_{\sigma(n)})$ and $d_j \leq d'_j$. Let T' be the tardiness associated with the sequence $(d_{\sigma(j+1)}, \ldots, d_{\sigma(n)})$. Let k be the integer such that $\sigma(k) = j$. Suppose we exchange the due dates $d_{\sigma(j)}$ and $d_{\sigma(k)}$ in the sequence $(d_{\sigma(j)}, \ldots, d_{\sigma(n)})$. As a result, the decrease of the total tardiness is equal to $\max(0, C_i - d_{\sigma(j)}) + \max(0, C_k - d_{\sigma(k)}) - \max(0, C_i - d_{\sigma(k)}) - \max(0, C_k - d_{\sigma(j)})$ which is non-negative as shown in the first part of the Proposition 86 because $C_i < C_k$ and $d_{\sigma(k)} \leq d_{\sigma(j)}$. Now the two sequences are identical up to index $j+1$. We iterate until the sequences are the same. Hence $lb_{Chu} \leq lb_1$. □

We provide an example for an instance with 3 activities, for which lb_1 is equal to 1 whereas lb_{Chu} is equal to 0 (see Table 11.4 and Figure 11.8). For this instance, lb_1 is the optimal total tardiness.

A_i	r_i	p_i	d_i
A_1	0	1	3
A_2	1	1	2
A_3	1	1	2

Table 11.4: Example with $lb_1 > lb_{Chu}$.

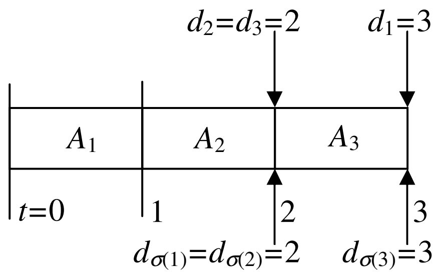


Figure 11.8: Assignment of Due Dates Obtained by the Algorithm Computing lb_1 (d_1, d_2, d_3) and by Chu's Algorithm $(d_{\sigma(1)}, d_{\sigma(2)}, d_{\sigma(3)})$.

Improving the Lower Bound

To improve our lower bound, we extensively use the Generalized Emmons Rules as a pre-processing step before the computation of the lower bound. The lower bound is valid in the preemptive case hence, we can apply the Generalized Emmons Rules. They allow us to tighten the release dates, which has a dramatic impact on the value of the lower bound.

Recall that, to compute our lower bound, we chronologically build a preemptive schedule (see Section 11.2.2). We still follow the same algorithm but, at each time point where a piece of activity to schedule next has to be chosen, we apply the Generalized Emmons Rules on the activities that are not scheduled yet.

Since there are $O(n)$ relevant time points and since the propagation of the Generalized Emmons Rules runs in $O(n^3)$, the improved lower bound can be computed in $O(n^4)$. In practice, the propagation of the Generalized Emmons Rules is much "faster" than $O(n^3)$ and the bound is computed in a reasonable amount of time. In the following we use the notation $lb_{\mathbf{2}}$ to refer to this bound.

11.2.3 Constraint Propagation

Focacci [102] has recently proposed an original approach based on Constraint Programming to compute a lower bound of $1|r_i|\sum T_i$. In this approach, each activity is associated with a constrained variable identifying all possible positions (first, second, third, *etc.*) that the activity can assume in a schedule. Following this idea, we present some rules that deduce that an activity cannot be executed in some positions. This information allows us to adjust the release dates.

From now on, we assume that activities are sorted in non-decreasing order of due dates.

As in Section 11.2.1, $[i]$ denotes the index of the activity which is completed in the i^{th} position in the SRPT schedule. We know that $C_{[i]}$ is a lower bound of the completion time of the activity scheduled in i^{th} position and according to Proposition 84, $\sum_{i=1}^{n} \max(C_{[i]} - d_i, 0)$ is a lower bound of the total tardiness (each activity A_i is assigned to the completion time $C_{[i]}$).

Suppose now that we want to compute a lower bound of the total tardiness under the hypothesis that A_i is scheduled in the k^{th} position. We first assign the completion time $C_{[k]}$ to A_i and we reassign all other completion times to all other activities as follows.

- If $k < i$ the activities $A_k, A_{k+1}, \ldots, A_{i-1}$ are assigned to $C_{[k+1]}, C_{[k+2]}, \ldots, C_{[i]}$ and the other assignments do not change.

- If $k > i$ the activities $A_{i+1}, A_{i+2}, \dots, A_k$ are assigned to $C_{[i]}, C_{[i+1]}, \dots, C_{[k-1]}$ and the other assignments do not change.

Following these new assignments, we have a new lower bound. If it is greater than *criterion* (the value of the objective function), then A_i cannot be in the k^{th} position.

Now assume that we have shown that positions $1, \dots, k$ are not possible for an activity A_i then A_i cannot be completed before $C_{[k+1]}$ and cannot start before $C_{[k]}$. Hence, the release date r_i can be adjusted to $\max(r_i, C_{[k+1]} - p_i, C_{[k]})$. Moreover, if $k > 1$ then i cannot be scheduled first. Of course, symmetric rules hold when it is known that the last k positions are not possible for A_i.

To implement this constraint propagation rule, we just have to use the $O(n \log n)$ algorithm of Chu [75], to compute the values $C_{[1]}, C_{[2]} \dots, C_{[n]}$. Then, for each activity A_i and for each position k, the lower-bound can be recomputed in linear time thanks to the reassignment rules provided above. This leads to an overall time complexity of $O(n^3)$.

Actually, this algorithm can be improved as follows. Assume that we have computed the lower bound under the assumption that A_i is scheduled in position k. To compute the lower bound under the assumption that A_i is scheduled in position $k-1$, we just have to exchange the assignments of activity i and activity $[k-1]$. The modification of the lower bound is then $\max(0, C_{[k-1]} - d_i) - \max(0, C_{[k]} - d_i) + \max(0, C_{[k]} - d_{[k-1]}) - \max(0, C_{[k-1]} - d_{[k-1]})$. Hence, we can "try" all possible positions for A_i in linear time. All impossible positions can thus be computed in $O(n^2)$.

11.3 Sum of Transition Times and Sum of Transition Costs

In this section we pay attention to constraint propagation of transition time and transition cost constraints. To simplify the presentation, we only consider the case where there are no cumulative resources. We do include the possible presence of alternative resources (see Section 7.2.4). We remark that much of the material in this section is based upon material in [103].

A comprehensive review of the research on scheduling involving setup considerations is given in [7]. The authors review the literature on scheduling problems with sequence-dependent and sequence-independent setup times, on a single machine and parallel machines. They finally suggest directions for future research in the field. Some of these directions receive attention in this section, *e.g.*, emphasis on multi-machine scheduling problems, on multi-criteria objectives, and on a generalized shop environment. An important reference for this section is the paper of Brucker and Thiele [51] where the authors propose a branch and bound algorithm for a scheduling problem with sequence-dependent setup times. We also mention [105, 102] that extend on this work in the context of CP. For more general considerations on cost-based constraint propagation we refer to [102].

The basis for the constraint propagation of transition times and transition costs described in this section is formed by using a *routing problem* as a relaxation of the scheduling problem. In this problem, one has a set of **start** nodes, a set of **internal** nodes, and a set of **end** nodes. Each internal node i represents an activity A_i. When having m alternative machines, one is looking for m disjoint *routes* or *paths* in the graph defined by these three sets. Each route corresponds to a different machine, starting in the start node of the machine, traversing a sequence of internal nodes, and ending in the end node of the machine. More precisely, let $I = \{1, \dots, n\}$ be a set of n nodes, and $E = \{n+1, \dots, n+m\}$ and $S = \{n+m+1, \dots, n+2*m\}$ two sets of m nodes. Nodes in I represent internal nodes, nodes in S represent start nodes, and nodes in E represent end nodes. A global constraint is defined ensuring that m different routes $\rho_1, \dots, \rho_m$ exist such that all internal nodes are visited exactly once by a route starting from a node in S and ending in a node in E. Start nodes $n+m+1, \dots, n+2*m$ belong to routes $\rho_1, \dots, \rho_m$, respectively. End nodes $n+1, \dots, n+m$ belong to routes $\rho_1, \dots, \rho_m$, respectively. Moreover, sets of possible routes can be associated to each internal node.

In the CP model three variables per node are defined. Variables $next_i$ and $prev_i$ identify the nodes visited directly after and directly before node i. Variables $route_i$ identify the route node i belongs to. Variables $next_i$ and $prev_i$ take their values in $\{1, \dots, n+2m\}$. Variables $route_i$ take their values in $\{1, \dots, m\}$. Each start and end node has its route variable bound, *i.e.*, $route_{n+1} = 1, \dots, route_{n+m} = m$, $route_{n+m+1} = 1$, $\dots$, $route_{n+2m} = m$. In order to have a uniform treatment of all nodes inside the constraint, each start node $n+m+u$ has its $prev_{n+m+u}$ variable bound to the corresponding end node ($prev_{n+m+u} = n+u$), and each end node $n+u$ has its $next_{n+u}$ variable bound to the corresponding start node ($next_{n+u} = n+m+u$). There furthermore exists a transition cost c^u_{ij} that expresses that if node j is visited directly after node i on a route u ($next_i = j, route_i = route_j = u$), a cost c^u_{ij} is induced. A feasible solution is defined as an assignment of distinct values to each next variable, while avoiding sub-tours (tours containing only internal nodes), and respecting the constraints

$$\begin{aligned} next_i = j \quad &\Leftrightarrow \quad prev_j = i \\ next_i = j \quad &\Rightarrow \quad route_i = route_j \end{aligned}$$

The problem is then to find an optimal feasible solution, *i.e.*, a feasible solution that minimizes

$$\sum_{i=1}^{n} c^u_{i\,next_i} \tag{11.7}$$

As said, the routing problem described constitutes a relaxation of the global scheduling problem. If an internal node i has its next variable assigned to another internal node j, activity A_i directly precedes activity A_j. If an internal node i has its next variable assigned to an ending node $n+u$, activity A_i is the last activity scheduled on machine M_u. The transition costs c^u_{ij} of the routing problem correspond to the setup times ($setup(A_i, A_j, M_u)$) or setup costs ($setupCost(A_i, A_j, M_u)$) between activities; therefore the minimization of the total transition cost (11.7) in the routing problem corresponds to the minimization of the sum of setup times or setup costs in the scheduling problem.

11.3.1 Route Optimization Constraint

In this section, we describe the constraint propagation given in [103]. One of the basic ideas is to create a global constraint having a propagation algorithm aimed at removing those assignments from variable domains which do not improve the best solution found so far. Domain reduction is achieved by optimally solving a problem which is a relaxation of the original problem.

We consider the Assignment Problem [82] as a relaxation of the routing problem described and thus also of the global scheduling problem. The Assignment Problem is the graph theory problem of finding a set of disjoint sub-tours such that all the vertices in a graph are visited and the overall cost is minimized.

In the routing problem we look for a set of m disjoint routes each of them starting from a start node and ending in the corresponding end node covering all nodes in a graph, *i.e.*, considering that each end node is connected to the corresponding start node, we look for a set of m disjoint tours each of them containing a start node. This problem can be formulated as an Assignment Problem on the graph defined by the set of nodes in the routing problem and the set of arcs (i, j) such that $j \in domain(next_i)$. The cost on arc (i, j) is the minimal setup cost (or time), *i.e.*,

$$\min_{u \in domain(route_i) \cap domain(route_j)} setupCost(A_i, A_j, M_u).$$

The value of the optimal solution of the Assignment Problem is obviously a lower bound on the value of the optimal solution of the routing problem. The primal-dual algorithm described in [65] provides an optimal integer solution for the Assignment Problem with a $O(n^3)$ time complexity. Besides this optimal assignment with the corresponding lower bound LB on the original problem, a reduced cost matrix $\bar{c}$ is obtained. Each $\bar{c}_{ij}$ estimates the additional cost to be added to LB if variable $next_i$ takes the value j. These results can be used both in constraint propagation as in the definition of search heuristics. The lower bound LB is trivially linked to the *criterion* variable representing the objective function through the constraint $LB \leq criterion$. More interesting is the propagation based on reduced costs. Given the reduced cost matrix $\bar{c}$, it is known that $LB_{next_i=j} = LB + \bar{c}_{ij}$ is a valid lower bound for the problem where $next_i$ takes the value j. Therefore we can impose

$$LB_{next_i=j} > ub(criterion) \Rightarrow \ next_i \neq j$$

As said, solving the Assignment Problem at the root can be done in $O(n^3)$. Each following assignment recomputation due to domain reduction can be done in $O(n^2)$ time though (see [65] for details). The reduced cost matrix is obtained without extra computational effort. Thus, the total time complexity of the constraint propagation algorithm is $O(n^2)$.

The following improvement of the use of the reduced costs is also exploited. We want to evaluate if value j could be removed from the domain of variable $next_i$ on the basis of its estimated cost. Let $next_i = k$ and $next_l = j$ in the optimal assignment. In order to assign $next_i = j$, l and k must be re-assigned. The exact cost of this re-assignment can be calculated in $O(n^2)$, thus increasing the global complexity of the constraint propagation algorithm. In [104], two bounds on this cost have been proposed, the calculation of which does not increase the total time complexity of the constraint propagation algorithm, which therefore remains $O(n^2)$. The events triggering this propagation are changes in the upper bound of the objective function variable *criterion* and each change in the domains of the *next*, *prev*, and *route* variables. Note that the assignment is recomputed only when the cost of an arc (i, j) that is part of the current solution increases its value over a certain threshold. The threshold T can be calculated as the minimum between the minimal reduced cost on row i and the minimal reduced cost on column j (excluding the zero reduced cost $\bar{c}_{ij}$).

$$T = \min(\min_{h \neq j}(\bar{c}_{ih}), \min_{k \neq i}(\bar{c}_{kj}))$$

Recomputing the assignment is needed every time the removed value j from $next_i$ belongs to the solution of the Assignment Problem (cost c_{ij} is set to infinite), and it may be needed when the reduction of the domain of $route_i$ or $route_j$ increases the minimal cost that is to be paid to go from i to j in any of the remaining possible routes. In all other cases no recomputation is needed since an increase in the cost of an arc that does not belong to the optimal solution does not change the optimal solution itself.

We remark that reduced cost fixing appears to be particularly suited for CP. In fact, while reduced cost fixing is extensively used in OR, it is usually not exploited to trigger other constraints, but only in the following lower bound computation, *i.e.*, the following node in the search tree. When embedded in a CP framework, the reduced cost fixing produces domain reductions which usually trigger propagation from other constraints in the problem through shared variables.

11.3.2 Precedence Graph Constraint

Linking the routing model and the scheduling model is done thanks to a *precedence graph constraint*. This constraint maintains for each machine M_u an extended precedence graph G_u that allows to represent and propagate temporal relations between pairs of activities on the machine as well as to dynamically compute the transitive closure of those relations. More precisely, G_u is a graph whose vertices are the alternative activities A_i^u that may execute on machine M_u (see Section 7.2.4). A node A_i^u is said to **surely contribute** if machine M_u is the only possible machine on which A_i can be processed. Otherwise, if activity A_i can also be processed on other machines, the node A_i^u is said to **possibly contribute**. Two kinds of edges are represented on G_u:

- A **precedence edge** between two alternative activities $A_i^u \rightarrow A_j^u$ means that if machine M_u is chosen for both activities A_i and A_j, then A_j will have to be processed after A_i on M_u.

- A **next edge** between two alternative activities $A_i^u \rightsquigarrow A_j^u$ means that if machine M_u is chosen for both activities A_i and A_j then A_j will have to be processed directly after A_i on M_u. No activity may be processed on M_u between A_i and A_j.

The first role of the precedence graph is to incrementally maintain the closure of this graph when new edges or vertices are inserted, *i.e.*, to deduce new edges given the ones already present in the graph. The following five rules [137] are used by the precedence graph:

1. If $A_i^u \rightarrow A_j^u$, $A_j^u \rightarrow A_i^u$, and A_i^u surely contributes then A_j^u does not contribute (Incompatibility rule).

2. If $A_i^u \rightarrow A_l^u$, $A_l^u \rightarrow A_j^u$, and A_l^u surely contributes then $A_i^u \rightarrow A_j^u$ (Transitive closure through contributor).

3. If $A_l^u \rightsquigarrow A_i^u$, $A_l^u \rightarrow A_j^u$, and A_l^u surely contributes then $A_i^u \rightarrow A_j^u$ (Next-edge closure on the left).

4. If $A_j^u \rightsquigarrow A_l^u$, $A_i^u \rightarrow A_l^u$, and A_l^u surely contributes then $A_i^u \rightarrow A_j^u$ (Next-edge closure on the right).

5. If for all A_l^u either $A_l^u \rightarrow A_i^u$ or $A_j^u \rightarrow A_l^u$ then $A_i^u \rightsquigarrow A_j^u$ (Next-edge finding).

New edges are added on the precedence graph G_u by the scheduling constraints (precedence and resource constraints) and by the route optimization constraint (whenever a variable $next_i$ is bound a new next-edge is added). Besides computing the incremental closure, the precedence graph also incrementally maintains the set of activities that are possibly next to a given activity A_i^u given the current topology of G_u. As such it allows to effectively reduce the domain of the variables $next_i$ and $prev_i$. Furthermore, the precedence graph constraint propagates the current set of precedence relations expressed on G_u on the start and end variables of activities.

11.4 Conclusion

In this chapter constraint propagation methods related to the minimization of the number of late jobs and to the minimization of setup times and setup costs have been presented. They drastically improve the behavior of Constraint-Based Scheduling systems on problems involving these criteria. However, there are several other criteria. In particular, the total weighted tardiness is widely used in the industry but until now poor results are obtained on this problem. This constitutes a very interesting research area.

Finally, users of Constraint-Based Scheduling tools often have to define their own criteria. A research challenge is to design generic lower-bounding techniques and constraint propagation algorithms that could work for any criterion.

Chapter 12

Resolution of Disjunctive Problems

Disjunctive problems are scheduling problems where all resources are machines (hence activities on the same machine cannot overlap in time, *i.e.*, they are in disjunction). These problems have been widely studied in the literature. In this chapter we address three problems: (1) the Job-Shop Problem where operations of jobs have to be processed in a given order by some specified machines, (2) the Open-Shop Problem where operations of the same job cannot overlap in time but can be executed in any order and (3) the Preemptive Job-Shop Problem.

12.1 Job-Shop Scheduling

The Job-Shop Scheduling Problem (JSSP) consists of n jobs that are to be executed using m machines. The benchmark problems used in our computational study are such that each job consists of m activities of given processing times to be executed in a specified order. Each activity requires a specified machine and each machine is required by a unique activity of each job. The JSSP is an NP-hard problem [108], known to be among the most difficult in this class. Many different methods have been tested. Most of them are branch and bound methods based upon a disjunctive graph [56], but other approaches have been used with some success as well: Mixed Integer Programming (MIP), genetic algorithms, simulated annealing, tabu search, *etc.*

First, the JSSP is an optimization problem. The goal is to determine a solution with minimal makespan and prove the optimality of the solution. The makespan is represented as an integer variable *criterion* constrained to be greater than or equal to the end of any activity. Several strategies can be considered to minimize the value of that variable, *e.g.*, iterate on the possible values, either from the lower bound of its domain up to the upper bound (until one solution is found), or from the upper bound down to the lower bound (determining each time whether there still is a solution). In our experiments, the dichotomizing algorithm of Section 7.2.5 is used.

12.1.1 Branching Scheme

A branching procedure with constraint propagation at each node of the search tree (including disjunctive constraint propagation and Edge-Finding as described in Section 8.1.2 and Section 8.1.3) is used to determine whether the problem with makespan at most D has a solution. We rely on the fact that in order to obtain a solution it is sufficient to order the jobs on the machines. So the basic structure of this algorithm is as follows:

1. Select a resource among the resources on which the activities are not fully ordered.
2. Select an activity to execute first among the unordered activities that require the chosen resource. Post the corresponding precedence constraints. Keep the other activities as alternatives

to be tried upon backtracking.

3. Iterate step 2 until all the activities that require the chosen resource are ordered.

4. Iterate steps 1 to 3 until all the activities on all resource are ordered.

Scheduling problems are generally such that resources are not equally loaded. Some resources are, over some periods of time, more relied upon than others. These resources are often called "critical". Their limited availability is a factor which prevents the reduction of a project cost or duration. It is, in general, very important to schedule critical resources first, in order to optimize the use of these resources without being bound by the schedule of other resources. A standard way to measure the criticality of a resource consists of comparing the "demand" for the resource to its availability ("supply") over a specific period of time. Here, for example, the time period under consideration may run from the earliest release date of the activities to order, to the largest deadline. As the capacity of the resource is 1 at all times, the supply over this time interval is the length of the interval. The demand over the interval is the sum of the processing times of the activities to execute. Criticality is then defined as the difference between demand and supply. The main interest of ordering all the activities of the critical resource first is that when all these activities are ordered, there is very little work left to do because constraint propagation has automatically ordered activities on the other resources.

The next step requires the selection of the activity to schedule first. We choose the activity with the earliest release date; in addition, the activity with the smallest latest start time is chosen when two or several activities share the same release date.

Variants of this branch and bound procedure have been widely used in the literature [19, 37, 35, 44, 62, 63, 67, 78, 163, 177, 173, 180, 185] and have shown to be very efficient. Instances of the JSSP with up to 10 jobs and 10 machines can be solved within a few minutes of CPU time.

12.1.2 Computational Results

Table 12.1 gives the results obtained with the simple algorithm described above on the ten 10*10 JSSP instances used in the computational study of [8]. The columns "BT" and "CPU" give the total number of backtracks and CPU time needed to find an optimal solution and prove its optimality. Columns "BT(pr)" and "CPU(pr)" give the number of backtracks and CPU time needed for the proof of optimality. All CPU times are given in seconds on a 200 MHz PC.

Instance	BT	CPU	BT(pr)	CPU(pr)
FT10	50908	297.9	7745	28.1
ABZ5	18463	43.2	6964	16.0
ABZ6	857	3.1	270	0.9
LA19	24154	57.8	5655	13.7
LA20	115114	228.3	24704	50.2
ORB1	14769	50.3	5357	18.5
ORB2	99665	298.6	30447	89.1
ORB3	273597	906.6	25556	92.5
ORB4	97509	367.7	22422	86.3
ORB5	9163	27.3	3316	9.6

Table 12.1: Results on ten 10*10 instances of the JSSP

This simple algorithm can easily be improved, as shown in the above references. In particular, recent algorithms incorporate "Shaving" techniques [63, 163, 180, 194] as described in the next section. Moreover, complex constraint propagation methods that take (partially) into account precedence relations between activities have shown to be extremely powerful. The best techniques available so far [179, 204, 205] allow to prove the optimality of all 10*10 instances in less than 30 seconds of CPU time.

12.2 Open-Shop Scheduling

The Open-Shop Scheduling Problem (OSSP) consists of a set of n jobs $\{J_1, \ldots, J_n\}$ that have to be scheduled on m parallel identical machines $M_1, \ldots, M_m$. A job consists of a set of operations with a given machine and processing time. Operations of the same job cannot overlap in time. The goal is to find a schedule with minimal makespan. The OSSP is NP-hard and has been shown to be very hard to solve in practice.

In our Constraint-Based Scheduling model, the Open-Shop Problem is modeled as a set of activities A_{ij} with given processing times p_{ij}. Each activity A_{ij} requires two resources representing the machine on which the activity is executed and the job J_i (since activities of the same job cannot overlap).

As for the JSSP, a minimal makespan solution can be found by a dichotomizing search. At each iteration, we solve the resulting decision problem, *i.e.*, determine a solution with makespan lower than or equal to some value D or prove that no such solution exists.

12.2.1 Branching Scheme

The same branching scheme as for the JSSP is used to solve each decision variant of the OSSP. On top of this, we use a simple **dominance relation** to reduce the search space. Consider a solution of the OSSP and let us define the symmetric counterpart of this schedule (start each activity A_{ij} at time $t'_{ij} = \mathit{makespan} - (t_{ij} + p_{ij})$, where t_{ij} is the starting time of activity A_{ij} on the initial schedule. It is easy to prove that the symmetric schedule is still valid. So we can arbitrary pick any activity A_{ij} and impose, *a priori*, that it executes mostly in the first half of the schedule:

$$\mathit{start}(A_{ij}) \leq \left\lfloor \frac{\mathit{makespan} - p_{ij}}{2} \right\rfloor$$

An interesting question is how to determine the activity on which this domain reduction is to be achieved. To estimate the impact of this domain reduction, it is "tried". For each activity A_{ij}, we compute the sum of the domain sizes of the variables of the problem when A_{ij} is constrained to execute mostly in the first half of the schedule. This is, we hope, a rough estimation of the remaining hardness of the problem. The activity that minimizes this sum is picked.

12.2.2 More and More Propagation: Shaving

Global operations also called "Shaving" have been used by Carlier and Pinson [63] and by Martin and Shmoys [163] to reduce the search space for the JSSP. This mechanism is also valid for any disjunctive problem (*e.g.*, Dorndorf, Pesch and Phan-Huy [87] show in their computational study of the OSSP that it can be of great practical interest). The basic idea is very simple. At each node of the search tree and for each activity A_i, the earliest date x_i at which the activity can be scheduled without triggering a contradiction is computed. This basically consists of

1. iteratively trying a start time for the activity A_i,

2. propagating the consequence of this decision thanks to the edge-finding bounding technique, and

3. verifying that no contradiction has been detected.

The earliest date x_i is of great interest since it can serve to adjust the release date of the activity. A dichotomizing procedure can be used to determine the date x_i. It decreases both the theoretical and the practical complexities of the algorithm.

Several extensions of this mechanism are proposed by Péridy [180]. The underlying idea is to impose a decision (*e.g.*, a starting time for a given activity) and to exploit the consequences of this decision in more or less complex algorithms to obtain global information on the instance.

12.2.3 Computational Results

On top of the dominance relation and of the classical disjunctive branching scheme, we also apply two propagation features used by Dorndorf, Pesch and Phan-Huy [87] in their computational study of the Open-Shop :

- **Not-First, Not-Last** propagation (*cf.*, Section 8.1.4) and
- **Shaving**, as defined in Section 12.2.2.

To evaluate the efficiency of these propagation techniques, the branch and bound procedure has been run on 132 instances of the OSSP (52 instances from [41] and 80 instances from [114]). These instances contain up to 100 jobs. Four variants of the algorithm have been tested:

- **STD.** Propagation includes the disjunctive constraint, Edge-Finding and the Not-First and Not-Last rules applied only to the maximal set of unordered activities.
- **SH.** Earliest start times and latest end times are shaved using the STD propagation.
- **NFNL.** Similar to STD but the Not-First and Not-Last rules are applied to all possible subsets.
- **SH+NFNL.** Both Shaving and Not-First Not-Last propagation are used.

Within a one hour time-limit, 122 instances can be solved by any of the procedures that use Shaving. These results compare well to more complex procedures such as [41]. Dorndorf, Pesch and Phan-Huy [87] report better computational results thanks to an improved version of the dichotomic search.

With STD and NFNL, 108 and 107 instances only are solved respectively. Figure 12.1 reports the number of instances solved (Nb Solved) within a given time limit (CPU) for the four different versions of the search procedure.

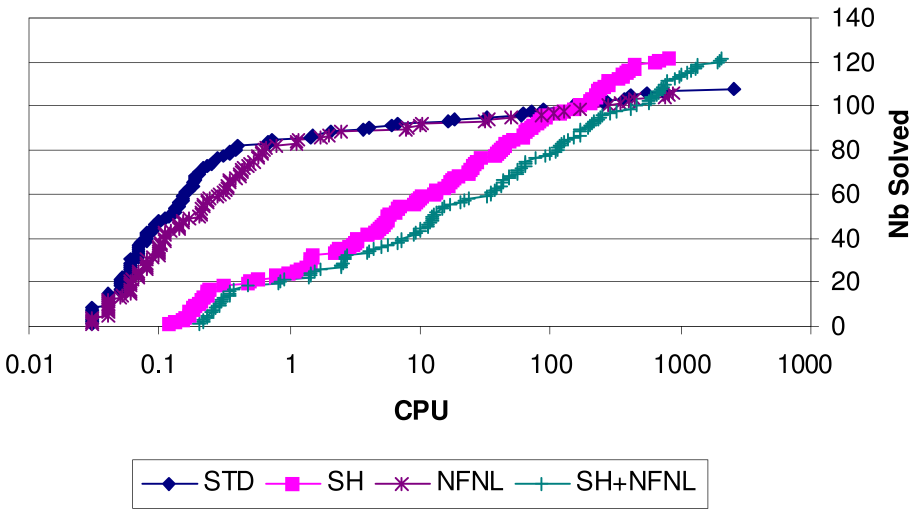


Figure 12.1: Number of OSSP instances solved in a given amount of CPU time

As noticed in [87], Shaving seems to be a key feature for solving the OSSP. However, it appears that the cost of applying systematically the NFNL rules is not balanced by the corresponding reduction of the search space. Applying the NFNL rules to the maximal subset can be done in linear time and seems to capture most of the propagation information.

12.2.4 Conclusion

As noticed by several researchers, the OSSP is harder to solve in practice than the JSSP. While all 10*10 instances of the JSSP are solved within a few second of CPU time, a 7*7 OSSP instance from [41] is still open!

Several techniques have been recently designed for the Open-Shop. In particular, we have not used, until now, any "intelligent backtracking" rule such as the one proposed by Guéret, Jussien and Prins [115]. We are quite confident that, together with powerful propagation techniques such as Shaving, this could lead to a drastic improvement for the resolution of the OSSP.

12.3 Preemptive Job-Shop Scheduling

The Preemptive Job-Shop Scheduling Problem (PJSSP) is the variant of the Job-Shop Scheduling Problem (JSSP) in which all activities are interruptible. Given are a set of jobs and a set of machines. Each job consists of a set of activities to be processed in a given order. Each activity is given an integer processing time and a machine on which it has to be processed. A machine can process at most one activity at a time. Activities may be interrupted at any time, an unlimited number of times. The problem is to find a schedule, *i.e.*, a set of integer execution times for each activity, that minimizes the makespan, *i.e.*, the time at which all activities are finished. We remark that the total number of interruptions of a given activity is bounded by its processing time minus 1. The PJSSP is NP-hard in the strong sense [108].

Formally, a start time variable $start(A_i)$, an end time variable $end(A_i)$, an integer processing time p_i and a set variable $set(A_i)$ are associated with each activity A_i. The set variable $set(A_i)$ represents the set of times at which A_i executes, $start(A_i)$ represents the time at which A_i starts, $end(A_i)$ the time at which A_i ends, and p_i the number of time units required for A_i. An additional variable, *makespan*, represents the makespan of the schedule. The following constraints apply:

- For every activity A_i, $p_i = |set(A_i)|$.
- For every activity A_i, $start(A_i) = \min_{t \in set(A_i)}(t)$ and $end(A_i) = \max_{t \in set(A_i)}(t+1)$. This implies $end(A_i) \geq start(A_i) + p_i$. We remark that in a given solution we have $end(A_i) = start(A_i) + p_i$ if and only if A_i is not interrupted.
- For every job $J = (A_1, A_2, \ldots, A_m)$ and every i in $\{1, 2, \ldots, m-1\}$, $end(A_i) \leq start(A_{i+1})$.
- For every machine M, if Ω_M denotes the set of activities to be processed on M, then, for every pair of activities (A_i, A_j) in Ω_M, $set(A_i)$ and $set(A_j)$ are disjoint.
- For every activity A_i, $0 \leq start(A_i)$.
- For every activity A_i, $end(A_i) \leq makespan$.

Constraint propagation techniques for all these types of constraints have been presented in Chapters 1 and 2.

As seen in Section 12.1, the most successful branch and bound approaches for the non-preemptive JSSP consist of ordering the set of activities Ω_M which require the same machine M. At each node, a machine M and a set $\Omega \subseteq \Omega_M$ are selected. For each activity A_i in Ω, a new branch is created where A_i is constrained to execute first (or last) among the activities in Ω. This decision is then propagated, through some variant or extension of the Edge-Finding constraint propagation technique. For the PJSSP, this branching scheme is not valid since activities are interruptible, and thus cannot just be ordered. This has a dramatic impact on the size of the search space. Let us consider, for example, 4 activities, each of processing time 10, to be executed on the same machine: there are $40!/(10!)^4 = 4705360871073570227520$ possible orders of the 40 activity time units in the PJSSP, to be compared with $4! = 24$ orders of the 4 activities in the JSSP. Considering only one of these activities A_i with earliest start time 0 and latest end time 40, the number of possible values for $set(A_i)$ is $40!/(10! * 30!) = 847660528$ in the PJSSP, to be compared with $40 - 10 = 30$ possible values for *start*(A_i) in the JSSP. As often in CP, the first and foremost thing to do (besides constraint propagation) to solve the PJSSP is to establish a dominance relation allowing the elimination of dominated solutions.

Section 12.3.1 presents such a dominance relation. Section 12.3.2 shows how it enables the definition of a simple depth first search algorithm for the Preemptive Job-Shop Scheduling Problem. Section 12.3.3 presents and compares different heuristic variants of this algorithm (still based on depth first search) and Section 12.3.4 examines the replacement of depth first search with various forms of limited discrepancy search [116].

12.3.1 Dominance Relation

The dominance relation introduced in this section allows the design of branching schemes which in a sense "order" the activities that require the same machine. The basic idea is that it does not make sense to let an activity A_i interrupt an activity A_j by which it was previously interrupted. In addition, A_i shall not interrupt A_j if the successor of A_i (in its job) starts after the successor of A_j. The following definitions and theorem provide a formal characterization of the dominance relation.

Definition 30. *For any schedule S and any activity A_i, we define the "due date of A_i in S" $d_S(A_i)$ as:*

- *the makespan of S if A_i is the last activity of its job;*
- *the start time of the successor of A_i (in its job) otherwise.*

Definition 31. *For any schedule S, an activity A_k has priority over an activity A_l in S ($A_k <_S A_l$) if and only if either $d_S(A_k) < d_S(A_l)$ or $d_S(A_k) = d_S(A_l)$ and $k \leq l$. Note that $<_S$ is a total order.*

Proposition 88. *For any schedule S, there exists a schedule $J(S)$ such that:*

1. *J(S) meets the due dates: $\forall A_i$, the end time of A_i in $J(S)$ is at most $d_S(A_i)$.*
2. *J(S) is "active": For any machine M and any time point t, if some activity $A_i \in \Omega_M$ is available at time t, M is not idle at time t (where "available" means that the predecessor of A_i is finished and A_i is not finished).*
3. *J(S) follows the $<_S$ priority order: $\forall M$, $\forall t$, $\forall A_k \in \Omega_M$, $\forall A_l \in \Omega_M$, $A_l \neq A_k$, if A_k executes at time t, either A_l is not available at time t or $A_k <_S A_l$.*

Proof. We construct $J(S)$ chronologically. At any time t and on any machine M, the available activity that is the smallest (according to the $<_S$ order) is scheduled. $J(S)$ satisfies properties 2 and 3 by construction. Let us suppose $J(S)$ does not satisfy property 1. Let A_i denote the smallest activity (according to $<_S$) such that the end time of A_i in $J(S)$ exceeds $d_S(A_i)$. We claim that:

- the schedule of A_i is not influenced by the activities A_k with $A_i <_S A_k$ (by construction);
- for every activity $A_k <_S A_i$, the time at which A_k becomes available in $J(S)$ does not exceed the time at which A_k starts in S (because the predecessor of A_k is smaller than A_i).

Let M be the machine on which A_i executes. The activities $A_k \in \Omega_M$ such that $A_k <_S A_i$ are, in $J(S)$, scheduled in accordance with Jackson's rule (*cf.*, [62]) applied to the due dates $d_S(A_k)$. Since $d_S(A_i)$ is not met, and since Jackson's rule is guaranteed to meet due dates whenever it is possible to do so, we deduce that it is *impossible* to schedule the activities $A_k \in \Omega_M$ such that $A_k <_S A_i$ between their start times in S and their due dates in S. This leads to a contradiction, since in S these activities *are* scheduled between their start times and their due dates. □

We call $J(S)$ the "Jackson derivation" of S. Since the makespan of $J(S)$ does not exceed the makespan of S, at least one optimal schedule is the Jackson derivation of another schedule. Thus, in the search for an optimal schedule, we can impose the characteristics of a Jackson derivation to the schedule under construction. This results in a significant reduction of the size of the search space.

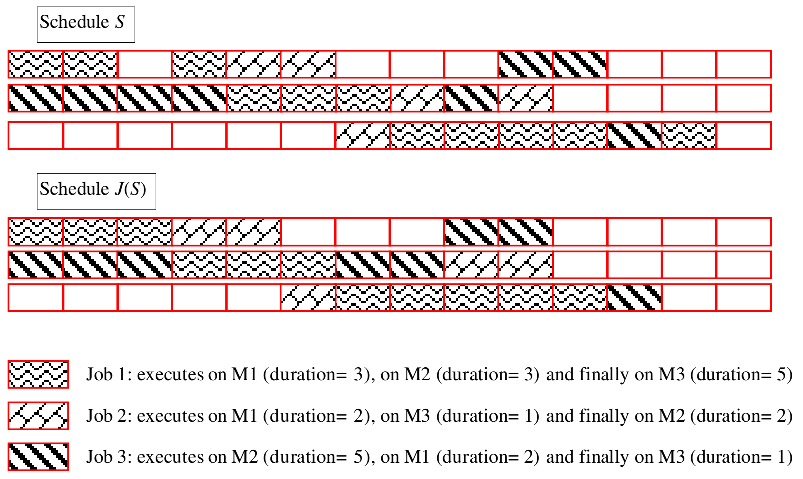

Schedule S
M1
M2
M3
Schedule $J(S)$
M1
M2
M3
Job 1: executes on M1 (duration= 3), on M2 (duration= 3) and finally on M3 (duration= 5)
Job 2: executes on M1 (duration= 2), on M3 (duration= 1) and finally on M2 (duration= 2)
Job 3: executes on M2 (duration= 5), on M1 (duration= 2) and finally on M3 (duration= 1)

Figure 12.2: A preemptive schedule and its Jackson derivation

12.3.2 Branching Scheme

The dominance relation of the previous section led us to develop the following branching scheme (which heavily exploits the dominance relation):

1. Let t be the earliest date such that there is an activity A_i available (and not scheduled yet) at t.
2. Compute K, the set of activities available at t on the same machine than A_i.
3. Compute *NDK*, the set of activities which are not "dominated" in K (as explained below).
4. Select an activity A_k in *NDK* (*e.g.*, the one with the smallest latest end time). Schedule A_k to execute at t. Propagate the decision and its consequences according to the dominance relation (as explained below). Keep the other activities of *NDK* as alternatives to be tried upon backtracking.
5. Iterate until all the activities are scheduled or until all alternatives have been tried.

Needless to say, the power of this branching scheme highly depends on the rules that are used to eliminate "dominated" activities in step 3 and propagate "consequences" of the choice of A_k in step 4. The dominance relation is exploited as follows:

- Whenever $A_k \in \Omega_M$ is chosen to execute at time t, it is set to execute either up to its earliest possible end time or up to the earliest possible start time of another activity $A_l \in \Omega_M$ which is not available at t.
- Whenever $A_k \in K$ is chosen to execute at time t, any other activity $A_l \in K$ can be constrained not to execute between t and the end of A_k. At times $t' > t$, this reduces the set of candidates for execution (A_l is "dominated" by A_k, hence not included in *NDK*). In step 4, "redundant" constraints can also be added: $\mathit{end}(A_k) + \mathit{rp}_t(A_l) \leq \mathit{end}(A_l)$, where $\mathit{rp}_t(A_l)$ is the remaining processing time of A_l at time t; $\mathit{end}(A_k) \leq \mathit{start}(A_l)$ if A_l is not started at time t.
- Let $A_k \in \Omega_M$ be the last activity of its job. Let $A_l \in \Omega_M$ be another activity such that either $l < k$ or A_l is not the last activity of its job. Then, if A_l is available at time t, A_k is not candidate for execution at time t (A_k is dominated by A_l).

The proof that these reductions of the search space do not eliminate all optimal schedules follows from the fact that $J(S)$ schedules are dominant. Indeed, in a $J(S)$ schedule, (i) an activity cannot be interrupted unless a new activity becomes available on the same resource, (ii) an activity A_k

cannot execute when another activity A_l is available, unless $A_k <_S A_l$, and (*iii*) we cannot have $A_k <_S A_l$ if A_k is the last activity of its job and either A_l is not the last activity of its job or $l < k$.

This branching scheme is integrated in the dichotomizing algorithm of Section 7.2.5. The resulting algorithm has been tested using different constraint propagation techniques for the resource constraints: Time-Tables (TT), Edge-Finding (EF) and network flows (SCF, AEC and GUTB). Computational results presented in [150] show that both Edge-Finding and the GUTB algorithm allow the resolution of difficult problems with 100 activities. In particular, five of the ten $10 * 10$ (*i.e.*, 10 machines $*$ 10 jobs $=$ 100 activities) instances used by Applegate and Cook [8] in their computational study of the (non-preemptive) Job-Shop Problem have been solved to optimality in a few hours of CPU time using GUTB. The Edge-Finding algorithm enabled the exact resolution of all of these ten instances with CPU time varying from 1 minute to 27 hours (3.4 hours on average) on a 200 MHz PC.

12.3.3 Heuristic Algorithms

Several variants of the previous algorithm can be designed to search for "good" rather than optimal solutions. Different algorithms shall then be compared not with respect to the time needed to solve the optimization problem (*i.e.*, find an optimal solution and prove it is optimal), but with respect to the value of the best solution found after a given amount of time. Several changes to the previous algorithm have been explored from this point of view.

Let us first consider the course of action to follow when a new solution has been found by the branch and bound algorithm. The alternative is either to "*continue*" the search for a better solution in the current search tree (with a new constraint stating that the makespan must be smaller than the current one) or to "*restart*" a brand new branch and bound search (with the same additional constraint). The main advantage of restarting the search is that the heuristic choices can rely on the result of the new propagation (based on the new upper bound), which shall lead to a better exploration of the search tree. The drawback is that parts of the new search tree may have been explored in a previous iteration, which results in redoing the same unfruitful work.

As far as the PJSSP is concerned, the restart strategy brings another point of flexibility, concerning the selection of an activity A_k in *NDK*. A basic strategy consists of selecting A_k according to a specific heuristic rule. Selecting the activity with the earliest deadline seems reasonable since it corresponds to the rule which optimally solves the Preemptive One-Machine Problem (see, for instance, [62]). However, one can also rely on the best schedule S computed so far: selecting the activity A_k with minimal $d_S(A_k)$ should help to find a better schedule when there exists one that is "close" to the previous one.

In addition, the Jackson derivation operator J and its symmetric counterpart K can be used to improve the current schedule. Whenever a new schedule S is found, derivations J and K can be applied to improve the current schedule prior to restarting the search. Several strategies can be considered, *e.g.*, apply only J, apply only K, apply a sequence of J's and K's. The following strategy appears to work well on average:

- compute $J(S)$ and $K(S)$;
- replace S with the best schedule among $J(S)$ and $K(S)$, if this schedule is strictly better than S ($J(S)$ is chosen if $J(S)$ and $K(S)$ have the same makespan);
- if S has been replaced by either $J(S)$ or $K(S)$, iterate.

Globally, this leads to five strategies based on the previous depth first search algorithm: DFS-C-E, DFS-R-E, DFS-R-E-JK, DFS-R-B and DFS-R-B-JK, where C, R, E, B, JK stand respectively for "*C*ontinue search in the same tree", "*R*estart search in a new tree", "select activities according to the *E*arliest deadline rule", "select activities according to their position on the *B*est schedule met so far" and "apply *JK* derivation operators".

Table 12.2 provides the results obtained on the preemptive variant of the ten $10 * 10$ instances used in [8]. Two constraint propagation techniques are considered: Time-Table as a fast algorithm that does not prune much, and Edge-Finding as a constraint propagation algorithm that requires more time but enables better pruning of the search tree. Each line of the table corresponds to a "constraint propagation + search" combination, and provides the mean relative error (MRE, in percentage) obtained after 1, 2, 3, 4, 5, and 10 min of CPU time. For each instance, the relative error is computed as the difference between the obtained makespan and the optimal value, divided by the optimal value. The MRE is the average relative error over the ten instances.

Table 12.3 provides the results obtained on the thirteen instances used by Vaessens, Aarts, and Lenstra [213] to compare local search algorithms for the non-preemptive Job-Shop Problem. As these instances differ in size, we allocated to each instance an amount of time proportional to the square of the number of activities in the instance. This means that column 1 corresponds to the allocation of 1 min to a $10 * 10$ problem, 15 sec for a $10 * 5$ problem, 4 min for a $20 * 10$ problem, *etc.* We used the square of the number of activities, because the time spent per node appeared to be approximately linear in the number of activities [151], and the number of decisions necessary to construct a schedule (*i.e.*, the depth of the search tree) is also proportional to the number of activities (hence the time necessary to reach the first solution tends to increase as the square of the number of activities). To our knowledge, five of the thirteen instances are, in the preemptive case, open (and we have been unable to solve them with the exact algorithm). For these instances, we applied a variant of Edge-Finding and Shaving (*cf.*, Section 12.2.2) to obtain a lower bound, and thus estimate the relative error.

Prop. algo.	Search strategy	1	2	3	4	5	10
Time-Table	DFS-C-E	16.74	16.37	16.25	16.25	16.25	16.18
	DFS-R-E	16.74	16.42	16.40	16.37	16.37	16.18
	DFS-R-E-JK	8.95	8.95	8.95	8.95	8.95	8.33
	DFS-R-B	14.67	14.48	14.48	14.48	14.13	13.72
	DFS-R-B-JK	8.32	8.16	7.74	7.73	7.73	7.34
Edge-Finding	DFS-C-E	5.23	4.64	3.80	3.09	2.94	1.55
	DFS-R-E	5.70	5.26	4.99	4.47	4.09	2.73
	DFS-R-E-JK	4.29	3.67	3.17	2.55	2.42	1.62
	DFS-R-B	4.23	3.68	3.41	2.82	2.80	1.41
	DFS-R-B-JK	1.69	1.32	0.86	0.80	0.79	0.65

Table 12.2: DFS results on the ten instances used in [8]

Prop. algo.	Search strategy	1	2	3	4	5	10
Time-Table	DFS-C-E	16.28	16.04	16.03	15.96	15.96	15.96
	DFS-R-E	16.34	16.08	16.06	16.05	16.04	16.02
	DFS-R-E-JK	9.22	9.22	9.22	9.22	9.22	9.04
	DFS-R-B	14.55	14.36	14.28	14.28	14.28	14.25
	DFS-R-B-JK	8.82	8.70	8.60	8.59	8.59	7.84
Edge-Finding	DFS-C-E	4.33	3.98	3.62	3.52	3.47	3.15
	DFS-R-E	4.99	4.80	4.49	4.25	3.96	3.72
	DFS-R-E-JK	4.02	3.74	3.32	3.26	3.22	3.03
	DFS-R-B	3.96	3.64	3.42	3.42	3.42	3.04
	DFS-R-B-JK	2.26	2.12	1.94	1.77	1.77	1.72

Table 12.3: DFS results on the thirteen instances used in [213]

These tables show that the use of the Edge-Finding technique enables the generation of good solutions in a limited amount of time and that the DFS-R-B-JK variant clearly outperforms the other algorithms, especially when the Edge-Finding technique is used.

12.3.4 Limited Discrepancy Search

Limited Discrepancy Search (LDS) [116] is an alternative to the classical depth first search algorithm. This technique relies on the intuition that heuristics make few mistakes through the search tree. Thus, considering the path from the root node of the tree to the first solution found by a DFS algorithm, there should be few "wrong turns" (*i.e.*, few nodes which were not immediately selected by the heuristic). The basic idea is to restrict the search to paths which do not diverge more than w times from the choices recommended by the heuristic. When $w = 0$, only the leftmost branch of the search tree is explored. When $w = 1$, the number of paths explored is linear in the depth of the search tree, since only one alternative turn is allowed for each path. Each time this limited search fails, w is incremented and the process is iterated, until either a solution is found or it is proven that there is no solution. It is easy to prove that when w gets large enough, LDS is complete. At each iteration, the branches where the discrepancies occur close to the root of the tree are explored first (which makes sense when the heuristics are more likely to make mistakes early in the search).

Several variants of the basic LDS algorithm can be considered:

- When the search tree is not binary, it can be considered that the ith best choice according to the heuristic corresponds either to 1 or to $(i - 1)$ discrepancies. In the following, we consider it represents $(i - 1)$ discrepancies because the second best choice is often much better than the third, *etc.* In practice, this makes the search tree equivalent to a binary tree where each decision consists of either retaining or eliminating the best activity according to the heuristic.

- The first iteration may correspond either to $w = 0$ or to $w = 1$. In the latter case, one can also modify the order in which nodes are explored during the first iteration (*i.e.*, start with discrepancies far from the root of the tree). The results reported below are based on a LDS algorithm which starts with $w = 0$.

- Korf proposed an improvement based on an upper bound on the depth of the search tree [131]. In our case, the depth of the search tree can vary a lot from a branch to another (even though it remains linear in the size of the problem), so we decided not to use Korf's variant. This implies that, to explore a complete tree, our implementation of LDS has a very high overhead over DFS.

- Walsh proposed another variant called "Depth-bounded Discrepancy Search" (DDS), in which any number of discrepancies is allowed, provided that all the discrepancies occur up to a given depth [216]. This variant is recommended when the heuristic is very unlikely to make mistakes in the middle and at the bottom of the search tree (*i.e.*, when almost all mistakes occur at low depth). We tried both LDS and DDS, and decided to focus on LDS, which appears to perform better. Some computational results with DDS are provided below.

Table 12.4 provides the results obtained by the five LDS variants, LDS-C-E, LDS-R-E, LDS-R-E-JK, LDS-R-B and LDS-R-B-JK, on the ten instances used by Applegate and Cook. Table 12.5 provides the results for the thirteen instances used by Vaessens, Aarts, and Lenstra. Figures 12.3 and 12.4 present the evolution of the mean relative error for the eight "constraint propagation + search" combinations in which the J and K operators are used. The combination of the Edge-Finding constraint propagation algorithm with LDS-R-B-JK appears to be the clear winner.

Tables 12.6 and 12.7 provide the results obtained using DFS and variants of LDS with the Edge-Finding constraint propagation algorithm and the R-E-JK and R-B-JK heuristic search strategies. In these tables, LDS^N and DDS^N refer to the application of the LDS and DDS principles on the N-ary tree, where each son of a node corresponds to scheduling one candidate activity on the corresponding machine. LDS and DDS (without the N) correspond to the application of LDS and

Prop. algo.	Search strategy	1	2	3	4	5	10
Time-Table	LDS-C-E	9.55	9.43	9.16	9.08	8.95	8.52
	LDS-R-E	10.46	9.87	9.22	8.98	8.98	8.52
	LDS-R-E-JK	7.68	6.75	6.75	6.75	6.75	5.98
	LDS-R-B	5.75	4.95	4.42	4.16	4.16	3.61
	LDS-R-B-JK	6.14	5.67	5.59	5.14	5.07	4.20
Edge-Finding	LDS-C-E	3.20	2.70	2.42	2.08	1.77	1.41
	LDS-R-E	3.52	2.90	2.67	2.39	2.25	1.66
	LDS-R-E-JK	2.17	2.03	1.86	1.71	1.38	1.24
	LDS-R-B	1.10	0.95	0.75	0.74	0.60	0.39
	LDS-R-B-JK	0.64	0.64	0.55	0.36	0.32	0.23

Table 12.4: LDS results on the ten instances used in [8]

Prop. algo.	Search strategy	1	2	3	4	5	10
Time-Table	LDS-C-E	11.43	11.12	11.08	10.89	10.53	10.43
	LDS-R-E	11.53	11.24	11.12	10.95	10.59	10.40
	LDS-R-E-JK	7.98	7.97	7.95	7.89	7.87	7.40
	LDS-R-B	6.58	5.92	5.64	5.44	5.26	4.68
	LDS-R-B-JK	5.93	5.82	5.78	5.66	5.66	4.60
Edge-Finding	LDS-C-E	3.57	3.02	2.85	2.77	2.57	2.27
	LDS-R-E	4.33	3.37	3.14	2.91	2.78	2.39
	LDS-R-E-JK	2.43	2.20	2.03	1.82	1.73	1.61
	LDS-R-B	2.25	1.80	1.76	1.74	1.58	1.13
	LDS-R-B-JK	1.75	1.28	1.03	0.92	0.92	0.79

Table 12.5: LDS results on the thirteen instances used in [213]

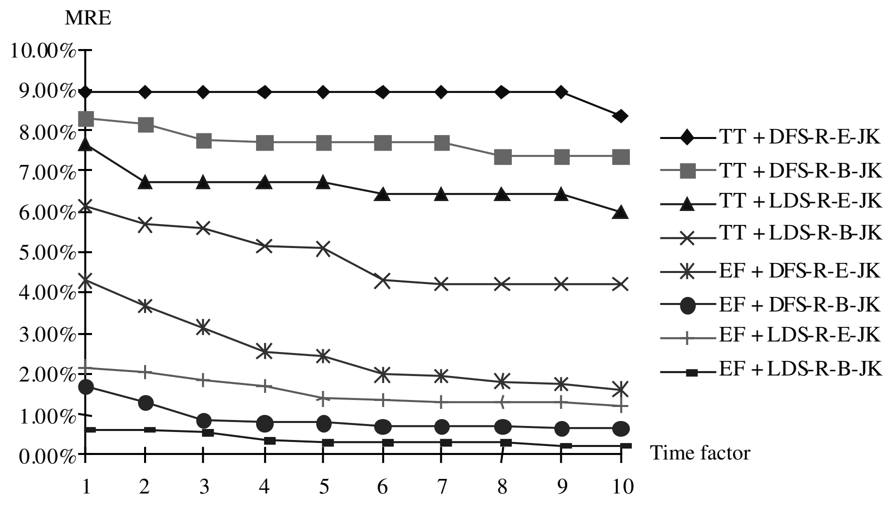


Figure 12.3: Evolution of the mean relative error on the ten instances used in [8]

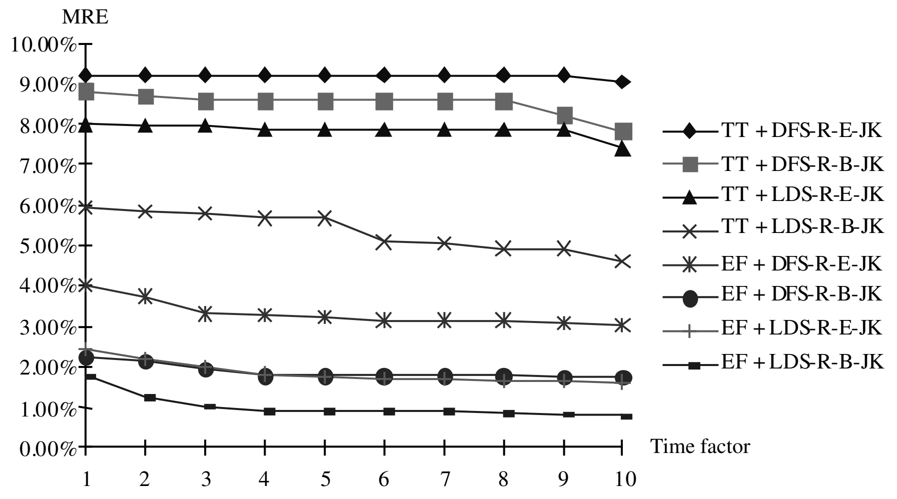


Figure 12.4: Evolution of the mean relative error on the thirteen instances used in [213]

DDS to the binary tree in which each decision consists of either scheduling or postponing the best candidate. Figure 12.5 illustrates the difference between the five algorithms, DFS, LDS, LDS^N, DDS, and DDS^N, on a ternary tree of depth three. “ITE” shows at which iteration each leaf of the tree is attained (starting with iteration number 0) and “ORD” shows in which order the leaves are visited. It shall be noted that, to explore a complete tree, LDS and DDS have a high overhead over DFS. Hence, LDS and DDS must quickly repair the bad decisions to beat DFS. LDS^N and DDS^N have a much smaller overhead on a complete tree. Hence, their performance is less dependent on the quick repair of the “worst” decisions. However, in LDS^N and DDS^N, the 2nd, 3rd, ... and Nth sons of a node are considered equal, which can lead to the early exploration of unpromising branches.

Prop. algo.	Search strategy	1	2	3	4	5	10
Edge-Finding	DFS-R-E-JK	4.29	3.67	3.17	2.55	2.42	1.62
	LDS-R-E-JK	2.17	2.03	1.86	1.71	1.38	1.24
	LDS^N-R-E-JK	2.66	2.13	1.93	1.74	1.62	1.27
	DDS-R-E-JK	2.81	2.20	1.93	1.71	1.67	1.34
	DDS^N-R-E-JK	3.17	2.50	2.33	2.25	1.63	1.51
	DFS-R-B-JK	1.69	1.32	0.86	0.80	0.79	0.65
	LDS-R-B-JK	0.64	0.64	0.55	0.36	0.32	0.23
	LDS^N-R-B-JK	1.10	0.91	0.70	0.64	0.53	0.47
	DDS-R-B-JK	2.01	1.68	1.29	1.11	0.89	0.72
	DDS^N-R-B-JK	1.98	1.63	1.34	1.21	1.03	0.81

Table 12.6: LDS variants on the ten instances used in [8]

LDS on the binary tree appears to be the best overall alternative. LDS^N-R-B-JK also performs well, in particular in Table 12.7. By contrast, DDS and DDS^N do not perform well. In particular, DDS-R-B-JK and DDS^N-R-B-JK do not perform better than DFS-R-B-JK in Table 12.6 and for the first values of the time factor in Table 12.7.

Edge-Finding appears as the most crucial of the techniques we used to provide good solutions to the PJSSP. On the 13 instances used in [213], all the algorithms that use Edge-Finding perform better than all the algorithms that do not. Tables 12.8 and 12.9 provide the average gains in MRE over the 8 DFS-R-$*$-$*$ and LDS-R-$*$-$*$ scenarios, when each of the other techniques is added. When Edge-Finding is not used, LDS appears as the second most important technique. When Edge-Finding is used, the contributions of the different techniques are closer one to the other. This confirms the observation in [30] that LDS appears to help the weaker algorithms to a greater extent than the

Prop. algo.	Search strategy	1	2	3	4	5	10
Edge-Finding	DFS-R-E-JK	4.02	3.74	3.32	3.26	3.22	3.03
	LDS-R-E-JK	2.43	2.20	2.03	1.82	1.73	1.61
	LDS^N-R-E-JK	2.61	2.32	2.21	1.93	1.86	1.81
	DDS-R-E-JK	3.46	3.10	2.98	2.30	2.29	2.16
	DDS^N-R-E-JK	3.69	2.96	2.90	2.79	2.61	2.39
	DFS-R-B-JK	2.26	2.12	1.94	1.77	1.77	1.72
	LDS-R-B-JK	1.75	1.28	1.03	0.92	0.92	0.79
	LDS^N-R-B-JK	1.50	0.99	0.87	0.86	0.86	0.78
	DDS-R-B-JK	2.68	2.47	2.20	1.56	1.51	1.26
	DDS^N-R-B-JK	2.68	2.40	1.87	1.59	1.56	1.32

Table 12.7: LDS variants on the thirteen instances used in [213]

stronger algorithms.

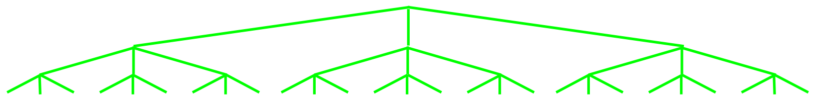
Prop. algo.	Search strategy	1	2	3	4	5	10
Time-Table	LDS	4.66	5.19	5.40	5.62	5.56	5.82
	B	2.24	2.18	2.27	2.38	2.49	2.53
	JK	4.13	4.05	3.87	3.85	3.79	4.04
Edge-Finding	LDS	2.12	1.85	1.65	1.36	1.39	0.72
	B	2.00	1.82	1.78	1.60	1.41	1.14
	JK	1.44	1.29	1.35	1.25	1.21	0.61

Table 12.8: Average gains on the ten instances used in [8]

Prop. algo.	Search strategy	1	2	3	4	5	10
Time-Table	LDS	4.23	4.35	4.41	4.55	4.69	5.01
	B	2.30	2.43	2.51	2.53	2.48	2.87
	JK	4.26	3.97	3.89	3.84	3.71	4.12
Edge-Finding	LDS	1.12	1.41	1.31	1.33	1.34	1.40
	B	1.39	1.32	1.20	1.10	1.00	1.02
	JK	1.26	1.07	1.12	1.13	1.03	0.78

Table 12.9: Average gains on the thirteen instances used in [213]

Globally, the best algorithm relies on LDS and on Edge-Finding, and provides excellent solutions to the Preemptive Job-Shop Scheduling Problem, in a reasonable amount of time.



ITE																											
DFS	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
LDS	0	1	2	1	2	3	2	3	4	1	2	3	2	3	4	3	4	5	2	3	4	3	4	5	4	5	6
LDS^{N}	0	1	1	1	2	2	1	2	2	1	2	2	2	3	3	2	3	3	1	2	2	2	3	3	2	3	3
DDS	0	3	4	2	4	5	3	4	5	1	4	5	3	5	6	4	5	6	2	4	5	3	5	6	4	5	6
DDS^{N}	0	3	3	2	3	3	2	3	3	1	3	3	2	3	3	2	3	3	1	3	3	2	3	3	2	3	3
ORD																											
DFS	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P	Q	R	S	T	U	V	W	X	Y	Z	&
LDS	A	D	J	C	I	Q	H	P	W	B	G	O	F	N	V	M	U	Z	E	L	T	K	S	Y	R	X	&
LDS^{N}	A	F	G	D	P	Q	E	R	S	B	J	K	H	T	U	I	V	W	C	N	O	L	X	Y	M	Z	&
DDS	A	E	I	C	J	P	F	K	Q	B	L	R	G	S	X	M	T	Y	D	N	U	H	V	Z	O	W	&
DDS^{N}	A	J	K	D	L	M	E	N	O	B	P	Q	F	R	S	G	T	U	C	V	W	H	X	Y	I	Z	&

Figure 12.5: The behavior of the five algorithms on a balanced ternary tree

Chapter 13

Cumulative Scheduling Problems

Many industrial scheduling problems are variants, extensions or restrictions of the "Resource-Constrained Project Scheduling Problem". Given (*i*) a set of resources with given capacities, (*ii*) a set of non-interruptible activities of given processing times, (*iii*) a network of precedence constraints between the activities, and (*iv*) for each activity and each resource the amount of the resource required by the activity over its execution, the goal of the RCPSP is to find a schedule meeting all the constraints whose makespan (*i.e.*, the time at which all activities are finished) is minimal. The decision variant of the RCPSP, *i.e.*, the problem of determining whether there exists a schedule of makespan smaller than a given deadline, is NP-hard in the strong sense [108].

In particular, the Hybrid Flow-Shop Scheduling Problem (HFSSP) is a special case of the RCPSP. To precisely define the Hybrid Flow-Shop Scheduling Problem, assume that a set $\{J_1, \ldots, J_n\}$ of n simultaneously available jobs must be sequentially processed on a set of s stages $S = \{1, 2, \ldots, s\}$. Each job is processed first at stage 1, then at stage 2, ..., and finally at stage s. At stage j, m_j identical parallel machines are available. Each job J_i can only be processed on one machine at a time and consists of s operations $O_{i1}, O_{i2}, \ldots, O_{is}$. An operation O_{ij} has a processing time p_{ij} and has to be processed without preemption on only one of the machines at stage j. The objective is, again, to find a schedule which minimizes the *maximum completion time*.

It is obvious that the HFSSP can be seen as a RCPSP where each operation is modeled by an activity and where each stage j is modeled by a resource of capacity m_j. Resource requirements are unitary, and the precedence network is made of n chains representing the jobs. Notice that if there are only two stages and if there is a single machine at each of the two stages, then the above Hybrid Flow-Shop Problem is solvable in $O(n \log n)$ time by the well-known algorithm developed by Johnson [127]. Other than this case, the Hybrid Flow-Shop Scheduling Problem is NP-hard in the strong sense even if there are only two available machines at one of the stages [121].

The aim of this computational study is to test the efficiency of the constraint propagation schemes described in the previous chapters and also to investigate one particular dimension along which problems differ. Within the cumulative scheduling class, we distinguish between highly disjunctive and highly cumulative problems: a scheduling problem is highly disjunctive when many pairs of activities cannot execute in parallel on the same resource; conversely, a scheduling problem is highly cumulative when many activities can execute in parallel on the same resource. To formalize this notion, we introduce the **disjunction ratio**, *i.e.*, the ratio between a lower bound of the number of pairs of activities which cannot execute in parallel and the overall number of pairs of distinct activities. A simple lower bound of the number of pairs of activities which cannot execute in parallel can be obtained by considering pairs $\{A_i, A_j\}$ such that either there is a chain of precedence constraints between A_i and A_j, or there is a resource constraint which is violated if A_i and A_j overlap in time. The disjunction ratio can be defined either globally (considering all the activities of a given problem instance) or for each resource R by limiting the pairs of activities to those that require at least one unit of R. The disjunction ratio of a resource of capacity 1 is equal to 1. The disjunction ratio of a cumulative resource varies between 0 and 1, depending on the precedence constraints and on the amounts of capacity that are required to execute the activities. In particular, the ratio is equal to 0 when there is no precedence constraint and no activity requires more than half of the resource capacity. When each stage is made of several machines, the disjunction ratio of the HFSSP is low due to the unitary resource requirements.

Needless to say, the disjunction ratio is only one of a variety of indicators that could be associated with scheduling problem instances. For example, the precedence ratio (also known as order strength [164] flexibility ratio, and density [85]), *i.e.*, the ratio between the number of pairs of activities which are ordered by precedence constraints and the overall number of pairs of distinct activities, is also important (a high precedence ratio makes the problem easier). Although some researchers, *e.g.*, [130], have worked on such indicators, we believe much more work is necessary to discover which ones are appropriate for designing, selecting, or adapting CP techniques with respect to the characteristics of a given problem.

In the following, we explore the hypothesis that the disjunction ratio is an important indicator of which techniques shall be applied to a cumulative scheduling problem. With this distinction in mind, we introduce several new techniques to solve the RCPSP. They will be also applied to the HFSSP.

Section 13.1 presents our general approach to the resolution of the RCPSP. Sections 13.2 and 13.3 respectively present some computational results on the HFSSP and on the RCPSP. They confirm that the techniques we use exhibit different behaviors on problems with different disjunction ratios.

13.1 General Framework

The aim of this section is to present a list (by no means exhaustive) of possible "ingredients" that can be incorporated in a CP approach to the RCPSP. We limit the discussion to the standard RCPSP. However, some of the techniques we propose also apply to extensions of the RCPSP, such as problems with interruptible activities.

First, the RCPSP is an optimization problem. The goal is to determine a solution with minimal makespan and prove the optimality of the solution. As for the Job-Shop, we represent the makespan as an integer variable constrained to be greater than or equal to the end of any activity. Again we use the dichotomizing algorithm of Section 7.2.5.

A branching procedure with constraint propagation at each node of the search tree is used to determine whether the problem with makespan at most D accepts a solution. As shown in the literature, there are many possible choices regarding the amount of constraint propagation that can be made at each node. [57, 83, 84] use simple bounding techniques compared to the more complex constraint propagation algorithms described in the previous chapters. Performing more constraint propagation serves two purposes: first, detect that a partial solution at a given node cannot be extended into a complete solution with makespan lower than or equal to D; second, reduce the domains of the start and end variables, thereby providing useful information on which variables are the most constrained. However, complex constraint propagation algorithms take time to execute, so the cost of these algorithms may not always be balanced by the subsequent reduction of search.

The deductive techniques for cumulative resources have been tested on the RCPSP and on the HFSSP. Computational results show that it is worth using such techniques when the disjunction ratio is low. Artificially adding "redundant" constraints, *i.e.*, constraints that do not change the set of solutions, but propagate in a different way, is another method for improving the effectiveness of constraint propagation. For example, [57, 58, 59] present branch and bound algorithms for the RCPSP which rely on the generation of redundant resource constraints. If S is a set of activities and m an integer value, and if for any subset s of S such that $|s| > m$, the activities of s cannot all overlap, then the following resource constraint can be added: "Each activity of S requires exactly one unit of a new (artificial) resource of capacity m".

Besides constraint propagation, a branching solution search procedure is also characterized by:

- **The types of decisions that are made at each node**. Most search procedures for the RCPSP chronologically build a schedule, from time 0 to time v. At a given time t, Demeulemeester and Herroelen [83] schedule a subset of the available activities; other subsets are tried upon backtracking. The main interest of this strategy is that some resource can be maximally used at time t, prior to proceed to a time $t' > t$. However, there may be many subsets to try upon backtracking, especially if the problem is highly cumulative. An example of non-chronological scheduling strategy is given by Carlier and Latapie [57]. Their strategy is based on dichotomizing the domains of the start variables: at each node, the lower or the upper half of the domain of a chosen variable V is removed and the decision is propagated. This strategy may work well if there are good reasons for selecting the variable V, rather than another variable (*e.g.*, when there is a clear resource bottleneck at a given time).

- **The heuristics that are used to select which possibilities to explore first**. When a chronological strategy is used, one can either try to "fill" the resources at time t (to avoid the insertion of resource idle time in the schedule) or select the most urgent activities among those that are available at time t. When a non-chronological strategy is used, the best is to focus first on identified bottlenecks.

- **The dominance relations that are applied to eliminate unpromising branches**. Several dominance rules have been developed for the RCPSP (see, for example [83, 22, 23]). These rules enable the reduction of the search to a limited number of nodes which "dominate" the others, *i.e.*, are guaranteed to include better schedules than the nodes that are eliminated. As for constraint propagation, dynamically applying complex dominance rules at each node of the search tree may prove more costly than beneficial.

- **The backtracking strategy that is applied upon failure**. Most CP tools rely on depth first chronological backtracking. However, "intelligent" backtracking strategies can also be applied to the RCPSP. For example, the cut-set dominance rule of Demeulemeester and Herroelen [83, 84] can be seen as an intelligent backtracking strategy, which consists of memorizing search states to avoid redoing the same work twice. When backtracking, the remaining subproblem is saved. In the remainder of the search tree, the algorithm checks if the remaining subproblem is not already proved unfeasible. The advantage of such techniques is that the identified impossible problem-solving situations are not encountered twice (or are immediately recognized as impossible). However, such techniques may require large amounts of memory to store the intermediate search results and, in some cases, significant time for their application.

We refer to [22] for a detailed study of all these aspects of the problem-solving strategy. Our aim is to evaluate the efficiency of the constraint propatation techniques described in the previous chapters. For this reason, we do not generate redundant resources, we do not apply the complex dominance rules of [83, 22, 23] and we do not use the cut-set rule. Hence the computational results could be improved.

Our solution search procedure is based upon [152]:

1. Initialize the set of **selectable** activities to the complete set of activities to schedule.

2. If all the activities have fixed start and end times, a solution is found, exit. Otherwise, remove from the set of selectable activities those activities which have fixed start and end times.

3. If the set of selectable activities is not empty, select an activity from the set (*e.g.*, one with smallest deadline), create a choice point for the selected activity (to allow backtracking) and schedule the selected activity from its earliest start time to its earliest end time. Then go to step 2.

4. If the set of selectable activities is empty, backtrack to the most recent choice point. (If there is no such choice point, report that there is no problem solution and exit.)

5. Upon backtracking, mark the activity that was scheduled at the considered choice point as not selectable as long as its earliest start and end times have not changed. Then go to step 2.

The correctness of this algorithm is easy to demonstrate. Indeed, the activity A_i chosen at step 3 must either start at its earliest start time (and consequently end at its earliest end time) or must be postponed to start later. But starting A_i later makes sense only if other activities prevent A_i from starting at its earliest start time, in which case the scheduling of these other activities must eventually result (thanks to constraint propagation!) in the update of the earliest start and end times of A_i. Hence a backtrack from step 4 signifies that all the activities which do not have fixed start and end times have been postponed. This is absurd as in any solution the activity which will start the earliest among those which have been postponed could be set to start at its earliest start time. This is why it is correct to backtrack from step 4 up to the most recent choice point.

Several constraint propagation algorithms can be associated with each resource. Among these algorithms, the Time-Table mechanism, is systematically applied. It guarantees that, at the end of the propagation, the earliest start time of each unscheduled activity is consistent with the start and end times of all the scheduled activities (*i.e.*, activities with bound start and end times).

13.2 Hybrid Flow-Shop Scheduling

The branch and bound procedure has been tested on 92 HFSSP instances generated by Carlier and Néron [59]. Problem size varies from 10 jobs and 5 stages, to 15 jobs and 5 stages. The machine configurations are equilibrated, *i.e.*, all stages have the same number of available machines (3 machines at all the stages), or are almost equilibrated (3 machines at all stages, excepted the third stage in which only 2 machines are available). As reported in [59] this makes the problem extremly hard to solve in practice.

The maximum CPU time allowed to solve a given problem instance was set at 3,600 seconds. Figure 13.1 reports the number of instances solved (Nb Solved) within a given time limit (CPU) for three different versions of our search procedure:

- with EnerGetic (EG) reasoning (Left-Shift / Right-Shift),
- with Edge-Finging (EF),
- with No (N) additional propagation.

On "easy" instances (those solved in a few seconds by all methods), the cost of the more complex time-bound adjustment algorithms is not balanced by the subsequent reduction of search, and the CPU time increases. On the contrary, when instances become hard, the energetic reasoning outperform all other techniques. Within a one hour time-limit, 76 instances are solved with energetic reasoning while 74 and 66 instances only are respectively solved with Edge-Finding and with Time-Tables only.

Over the 66 instances that have been solved by all three versions of the branch and bound (EG, EF and N), the reduction of the number of backtracks is very impressive. An average of 14917 backtracks was required by the EG version of the algorithm while EF and N required respectively 150853 and 1285595 backtracks! Following these results an improved version of these algorithms, incorporatig Shaving (see Section 12.2.2) and dominance rules, has been developped in [171].

13.3 Resource-Constrained Project Scheduling

The three versions of the algorithm were tested on two sets of data.

- The "KSD" set of Kolisch, Sprecher and Drexel [130], which includes 480 instances with 30 activities and 4 resources. These instances are interesting because they are classified according to various indicators, including the "resource strength," *i.e.*, the resource capacity, normalized so that the "strength" is 0 when for each resource R, $cap(R) = \max_i(cap(A_i, R))$, and the "strength" is 1 when scheduling each activity at its earliest start time (ignoring resource constraints) results in a schedule that satisfies resource constraints as well.

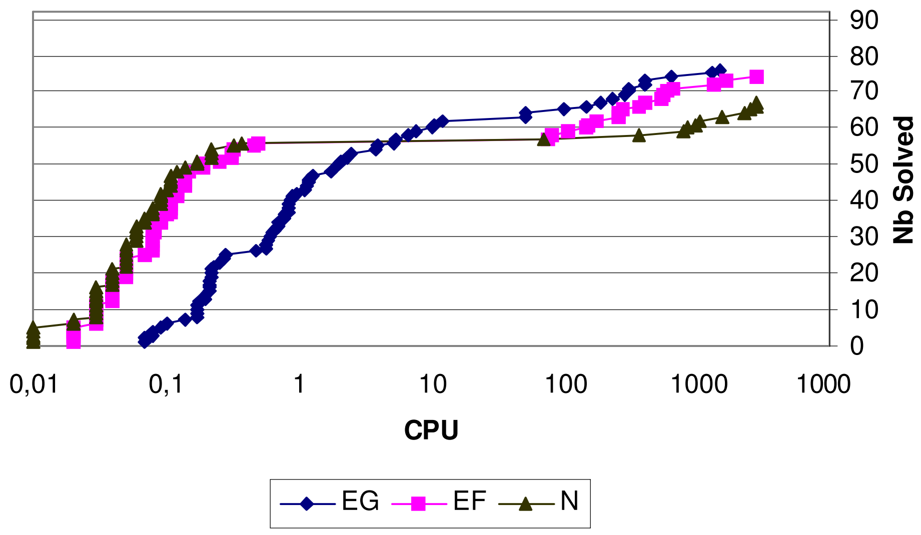


Figure 13.1: Number of HFSSP instances solved in a given amount of CPU time

- The "BL" set of Baptiste and Le Pape [22] which includes 40 instances with either 20 or 25 activities, and 3 resources. Each activity requires the 3 resources, with a required capacity randomly chosen between 0 and 60% of the resource capacity. 15 precedence constraints were randomly generated for problems with 20 activities; 45 precedence constraints were generated for problems with 25 activities.

The average disjunction ratios for the two sets of instances are respectively 0.56 for the KSD set and 0.33 for the BL set. As a matter of fact, the BL instances have been generated because it appeared that most of the classical instances from the literature are highly disjunctive (which is not representative of the real-life RCPSP instances that we observed).

Figures 13.2 and 13.3 provide the results obtained on the KSD and the BL sets of benchmarks with three different versions of our search procedure: with EnerGetic (EG) reasoning, with Edge-Finding (EF), with No (N) additional propagation. Each curve shows the number of instances solved in a given amount of CPU time.

The effect of the different satisfiability tests and time-bound adjustment algorithms clearly depends on the set of instances. Considering only the instances solved by all versions of the branch and bound (39 over 40 for the BL set and 353 over 480 for the KSD set), the reduction in the average number of backtracks between N and EG is equal to 97% and 24% on the BL and KSD sets respectively. On the KSD sets, the cost of the more complex time-bound adjustment algorithms is not balanced by the subsequent reduction of search, and the CPU time increases. On the contrary, EG performs much better than N on the BL set.

13.4 Conclusion

We have empirically evaluated several propagation techniques for cumulative resources. These techniques can be used not only for standard scheduling problems; some of them can also be used for preemptive scheduling problems (*e.g.*, partially elastic relaxation as defined in Section 9.2.3 or for fully elastic problems (*e.g.*, fully elastic relaxation as defined in Section 9.1). Propagation proves to be effective on some, but not all problem instances in the cumulative scheduling class. Computational results have shown that, on "highly disjunctive" project scheduling instances, the more complex constraint propagation algorithms induce an overhead that is not balanced by the resulting reduction of search. On the other hand, the most expensive techniques prove to be highly useful for the resolution of less highly disjunctive problems.

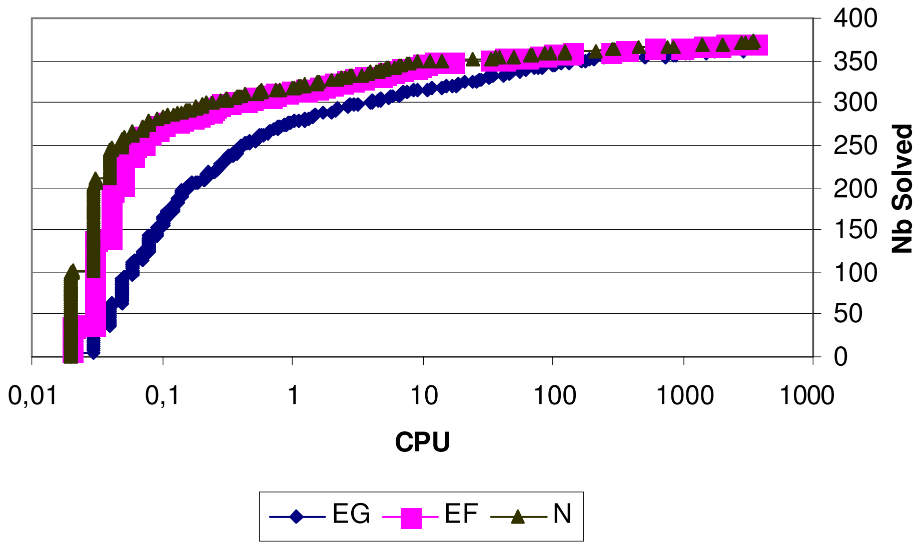


Figure 13.2: Number of KSD instances solved in a given amount of CPU time

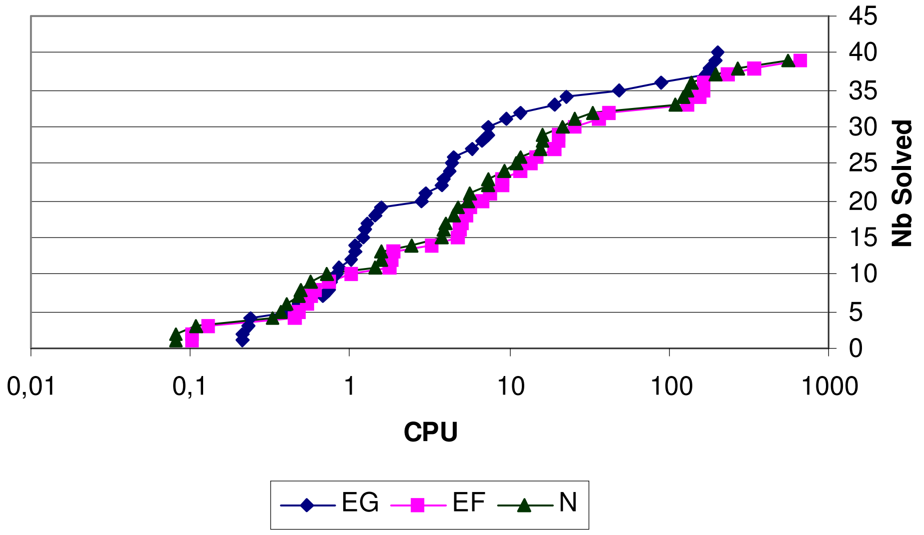


Figure 13.3: Number of BL instances solved in a given amount of CPU time

Let us remark that the branch and bound procedure does not incorporate all the results obtained by other researchers for the Resource-Constrained Project Scheduling Problem (RCPSP). In particular, we have not used, until now, any “intelligent backtracking” rule such as the cut-set rule of Demeulemeester and Herroelen [83]. This may seem a little “strange” given the excellent results reported in [84], in particular on the KSD instances, even with a limited use of the cut-set rule. It appears that many industrial scheduling problems include several additional features (including, for example, elastic activities) which seem to require the use of other techniques. Yet in the case of the pure RCPSP, it would be interesting to determine how subsequent improvements of the procedure would influence computational results, and hence our conclusions on the usefulness of the various propagation techniques.

Recently, several new lower bounds based on Linear Programming (LP) formulations of a relaxed RCPSP have shown to be very accurate [46, 166]. However, the size of the models used is so large that, even with complex column generation techniques, hours of CPU time are sometimes required to get a lower bound. Recently, Carlier and Néron [60] have shown that for each resource, an efficient LP-based lower bound can be tabulated, for each fixed value of the resource capacity. Thus, the computation of the bounds is not time consuming! This is, we think, one of the most promising research direction for the next few years.

Chapter 14

Min-Sum Scheduling Problems

We illustrate and compare the efficiency of the constraint propagation algorithms described in previous chapters on two well-known scheduling problems. In Sections 14.1 and 14.2 we respectively study the problems of minimizing the weighted number of late jobs on m parallel identical machines and the total tardiness on 1 machine. In Section 14.3 we consider a general-shop scheduling problem with sequence-dependent setup times and alternative machines where the optimization criteria are both makespan and sum of setup times.

14.1 Minimizing the Weighted Number of Late Jobs

The problem $P|r_i|\sum w_i U_i$ is very close to the resource constraint of Section 11.1. An instance of the decision-variant of this problem consists of a set of jobs $\{J_1, \ldots, J_n\}$, a set of m identical parallel machines, and an integer W. Each job J_i is described by a release date r_i, a due date d_i, a processing time p_i, and a weight w_i. The problem is to find an assignment of start times to jobs such that:

- less than m jobs are scheduled at each time point,
- each job starts after its release date,
- the weighted number of jobs that end after their due date is lower than or equal to W.

A job scheduled between its release date and its due date is "on-time". Conversely, a job that ends after its due date is "late". Even for the simple unweighted version of the problem $(1|r_i|\sum U_i)$, few exact approaches have been proposed.

- An efficient heuristic and a branch and bound procedure have been proposed in [79]. It is shown that the problem can be modeled by a Mixed Integer Program (MIP). Unfortunately, instances with more than 10 jobs could not be considered because of the size of the MIP.
- A branch and bound scheme based upon a "master sequence", *i.e.*, a sequence containing at least an optimal sequence, is proposed in [80]. Lower bounds are obtained by relaxing the release and the due dates according to the Kise, Ibaraki and Mine conditions [129].
- Another branch and bound procedure is presented in [181] where a lower bound is obtained by a Lagrangian relaxation of a MIP formulation.

The last three methods are in the same range of efficiency (95% of instances with 80 jobs can be optimally solved in a reasonable amount of time). Obviously, this problem fits our Constraint-Based Scheduling model. The m parallel machines are modeled by a resource of capacity m and each job J_i is modeled by an activity A_i that requires one unit of the resource. The initial domain of the start and end variables are set respectively to $[r_i, ub - p_i]$ and to $[r_i + p_i, ub]$, ub being an upper bound of the completion time of the jobs (*e.g.*, ub can be set as the maximum release date plus the sum of all processing times). The domain of the criterion variable *criterion* is set to $[0, W]$. Finally, the cost function is the sum of n functions $f_i(\mathit{end}(A_i))$ that equal 0 if $\mathit{end}(A_i) \leq d_i$ and w_i otherwise.

Let us remark that, with this model, $\bar{d}_i$ remains arbitrarily large (equal to ub) as long as it is not decided that J_i must be on-time.

14.1.1 Dominance Relations

Dominance relations allow to reduce the search space to schedules that have a particular structure. The most important dominance relation relies on the idea that it is better to schedule small and heavy jobs with large time windows than large and light jobs with small time windows. We also use two other dominance rules that respectively fix the start times of some jobs and split the problem into distinct subproblems.

Dominance of Small Jobs with Large Time-Windows

We rely on the observation that on any solution, if a large job J_j is on-time and is scheduled inside the time window $[r_i, d_i]$ of a smaller job J_i that is late, and if J_i is heavier than J_j, the jobs J_i and J_j can be "exchanged", *i.e.*, J_i becomes on-time and J_j becomes late. More formally our dominance rule is based upon the binary relation "$\prec$".

Definition 32. *For any pair of jobs J_i, J_j, $J_i \prec J_j$ if and only if*

$$\begin{cases} (p_i < p_j) \vee (p_i = p_j \wedge i < j) \\ w_i \geq w_j \\ r_i + p_i \leq r_j + p_j \\ d_j - p_j \leq d_i - p_i \end{cases}$$

The relation "$\prec$" is transitive and $\forall J_i, \forall J_j, J_i \prec J_j \Rightarrow J_i \neq J_j$. Thus, it defines a strict partial order on jobs. Proposition 89 is the theoretical basis of our dominance rule.

Proposition 89. *There is an optimal schedule such that for any pair of jobs $\{J_i, J_j\}$, if $J_i \prec J_j$ and J_j is on-time, then J_i is also on-time, i.e.,*

$$\forall J_i, \forall J_j, \neg[(J_i \prec J_j) \wedge (\mathit{end}(J_j) \leq d_j) \wedge (\mathit{end}(J_i) > d_i)] \tag{14.1}$$

Proof. Consider an optimal schedule $\mathcal{Q}$ such that the first index i for which there exists a job J_j that violates (14.1) is maximum. We have $(J_i \prec J_j) \wedge (\mathit{end}(J_j) \leq d_j) \wedge (\mathit{end}(J_i) > d_i)$. We build a new schedule obtained by "exchanging" J_i and J_j. More precisely, J_i is scheduled at the maximum of r_i and of the start time of J_j in $\mathcal{Q}$. J_j is scheduled after all other jobs (it then becomes late). It is obvious to verify that the new schedule is still feasible and optimal. Moreover, J_i is now on-time. Now, suppose that there exists a late job J_k such that $J_k \prec J_i$. We then have $J_k \prec J_i \prec J_j$. Moreover, J_k was also late on the initial schedule. Consequently, $k > i$ because of the choice of i. This contradicts our hypothesis on the choice of the initial schedule. □

Remark that the constraint in Proposition 89 still holds if $r_j + p_j$ is replaced by $\mathit{end}(J_j)$ and if $d_j - p_j$ is replaced by $\mathit{start}(J_j)$. Arc-consistency is achieved on this new constraint. It allows to prove that some jobs must be late or on-time and it tightens the domains of the start variables.

Decomposition Rule

The basic idea of the decomposition is to detect some cases in which the problem can be split into two subproblems. Each of them being solved independently.

Proposition 90. *Let t_1 be a time point such that for any non-late job (i.e., $\mathit{eet}_i \leq d_i$), either $\min(\bar{d}_i, d_i) \leq t_1$ or $t_1 \leq r_i$. Any optimal schedule, is the "sum" of an optimal schedule of $\{J_i : \mathit{eet}_i \leq d_i \wedge \min(\bar{d}_i, d_i) \leq t_1\}$ and of an optimal schedule of $\{J_i : \mathit{eet}_i \leq d_i \wedge t_1 \leq r_i\}$.*

Proof. Recall that late jobs can be scheduled arbitrarily late and we only consider on-time jobs. On-time jobs are either in $\{J_i : eet_i \leq d_i \wedge \min(\bar{d}_i, d_i) \leq t_1\}$ or in $\{J_i : eet_i \leq d_i \wedge t_1 \leq r_i\}$ and they do not compete for the machine. □

We only consider the values of t_1 that are earliest start times (if the problem can be decomposed at time t_1, it is easy to verify that it can also be decomposed at the first r_i after t_1). There are $O(n)$ distinct earliest start times and the decomposition test (at a given time point) obviously runs in linear time. Consequently, the overall decomposition test runs in $O(n^2)$. Once the problem has been decomposed, the optimum of each subproblem is computed and we simply have to verify that the sum of these optima is lower than or equal to W.

Straight Scheduling Rule

We use a simple dominance rule which schedules automatically a set of jobs if they "fit" in a particular interval.

Proposition 91. *Let $[t_1, t_2)$, be any time interval. Let $J(t_1, t_2)$ be the jobs, among those that are not late yet (i.e., $eet_i \leq d_i$), that may execute (even partially) in $[t_1, t_2)$ if they are on-time (i.e., $t_1 < \min(\bar{d}_i, d_i) \wedge t_2 > r_i$). Moreover, suppose that there exists a feasible schedule $\mathcal{Q}$ of $J(t_1, t_2)$ that is idle before t_1 and after t_2 on which all jobs are on-time. Then there exists an optimal overall schedule $\mathcal{R}$ of $\{J_1, \ldots, J_n\}$ such that between t_1 and t_2 the schedules $\mathcal{Q}$ and $\mathcal{R}$ are identical.*

Proof. Obvious. □

Consider now any time point t_1 and let $J(t_1)$ be the set of jobs J_i that do not have to be late and that can end after t_1 ($t_1 < \min(\bar{d}_i, d_i)$). We build a schedule of $J(t_1)$ as follows: Each time a job is completed, the available job with the smallest due date $\min(\bar{d}_i, d_i)$ is scheduled. Let t_2 be the current time point throughout the construction of the schedule:

1. if a job is completed after its due date then t_1 is not a valid candidate for straight scheduling
2. if $J(t_2)$ becomes empty then we have built a valid schedule of $J(t_1, t_2)$ where all jobs are on-time.

Now remark that if t_1 is not in $\{\min(\bar{d}_1, d_1), \min(\bar{d}_2, d_2), \ldots, \min(\bar{d}_n, d_n)\}$ then $J(t_1 - 1, t_2) = J(t_1, t_2)$ and a schedule that can fit in $[t_1, t_2)$ can also fit in $[t_1 - 1, t_2)$. Hence, the "straight scheduling rule" is applied for $t_1 = \min_i(r_i)$ and for $t_1 \in \{\min(\bar{d}_1, d_1), \min(\bar{d}_2, d_2), \ldots, \min(\bar{d}_n, d_n)\}$. In the following experiments, this rule is active only if the machine capacity is 1.

14.1.2 Branching Scheme

We use the dichotomizing algorithm of Section 7.2.5. The critical part is the resolution of the decision problem. The search tree is built as follows: While there are some jobs that can be late or on-time (*i.e.*, $eet_i \leq d_i < \bar{d}_i$),

1. select a job J_i such that $eet_i \leq d_i < \bar{d}_i$
2. constrain J_i to be on-time, *i.e.*, $end(J_i) \leq d_i$ (if a backtrack occurs, J_i must be late)
3. apply dominance rules and propagate constraints
4. check that there exists a feasible schedule of the jobs, *i.e.*, a schedule where all jobs are scheduled in their time-window, (if not, the problem is inconsistent and a backtrack occurs).

When the branch and bound succeeds, it has been decided for each job whether it is on-time or not. On top of that the weighted number of late jobs equals *criterion* (because arc-B-consistency is enforced on the criterion constraint). Moreover there is a feasible schedule of the jobs (step 4). Let us detail the heuristic used for the job selection and the procedure that checks whether there is a feasible schedule of the jobs that must be on-time.

Job Selection Heuristic

Let U be the set of jobs J_i that can be late or on-time (*i.e.*, $eet_i \leq d_i < \bar{d}_i$). Let $\max_i(w_i/p_i)$ be the maximum value of w_i/p_i among jobs in U and finally, let S be the subset of the jobs J_j in U such that $w_j/p_j > 0.9 * \max_i(w_i/p_i)$. Among jobs in S, we select a job whose time window, if it becomes on-time, *i.e.*, $[r_i, d_i]$, is the largest. This heuristic "bets" that it is better to schedule small and heavy jobs with large time windows rather than large and light jobs with tight time windows.

Feasibility Check

The aim of this step is to determine if there is a feasible schedule of the jobs that meets all time-windows. Even on a single machine, this problem is NP-hard in the strong sense. A large amount of work has been carried on this problem because it serves as the basis for the resolution of several disjunctive scheduling problems. The branch and bound algorithm of Carlier [54] has shown to be extremely efficient for this problem. Very large instances can be solved in a few amount of time. Hence, it is used as soon as the number of machines equals 1. Unfortunately, there is no such efficient algorithm in the cumulative case. Hence we use a procedure similar to the one described in Section 13.1 for the Resource-Constrained Project Scheduling Problem (in our case, there is no precedence constraint and resource requirements are unitary). The Left-Shift / Right-Shift propagation technique is used at this point (*cf.*, Section 9.3.6).

14.1.3 Computational Results

In this section we compare the efficiency of the constraint propagation algorithms described in previous chapters on the problem of minimizing the weighted number of late jobs. Several instances of $P|r_j|\sum w_j U_j$ have been generated following the generation scheme of [24, 17]. We pay attention to four important characteristics:

- The distribution of the processing times.
- The distribution of the weights.
- The overall "load" of the machines; where the load is the ratio between the sum of the processing times of the jobs and $m * (\max_i d_i - \min_i r_i)$.
- The margin $m_i = d_i - r_i - p_i$ of each job.

Our generation scheme is based on 6 parameters: the number of jobs n, the number of machines m, and the four statistical laws followed by the processing times, the weights, the release dates, and the margins (given r_i, p_i and m_i, the due date can be immediately computed $d_i = m_i + r_i + p_i$).

- Processing times are generated from the uniform distribution over $[p_{\min}, p_{\max}]$.
- Weights are generated from the uniform distribution $[1, w_{\max}]$.
- Release dates are generated from the normal distribution $(0, \sigma)$.
- Margins are generated from the uniform distribution $[0, m_{\max}]$.

Given these parameters and relying on the fact that most of the release dates are in $[-2\sigma, 2\sigma]$, the *load* is approximately equal to:

$$load = n\frac{p_{\min} + p_{\max}}{2m(4\sigma + p_{\max} + m_{\max})}$$

Given $n, p_{\min}, p_{max}, m_{\max}$ and *load*, this allows us to determine a correct value of σ. One instance has been generated for each of the combinations of the parameters $(n, m, (p_{\min}, p_{max}), m_{\max}, load, w_{\max})$. See Table 14.1. Tables 14.2, 14.3, and 14.4 report the results obtained for the three different values of m. Each line summarizes the results obtained on a set of 90 instances of the same size n. The

Parameter	Range
n	$\{10, 20, 30, 40, 50\}$
m	$\{1, 3, 6\}$
$(p_{\min}, p_{\max})$	$\{(0, 100), (25, 75)\}$
$m_{\max}$	$\{50, 200, 350, 500, 650\}$
load	$\{1.0, 1.6, 2.2\}$
$w_{\max}$	$\{1, 10, 100\}$

Table 14.1: Instances generation parameters

column "% solved" provides the ratio of instances that have been solved within 600 seconds of CPU time on a PC HP Vectra 350 MHz running Windows 95. The columns "Avg. CPU" and "Avg. BCK" report respectively the average CPU time in seconds and the average number of backtracks required to solve the instances. The column "Max CPU" reports the maximum CPU time. Instances that could not be solved within the time limit are not taken into account in these computations.

n	% solved	Avg. CPU	Max CPU	Avg. BCK
10	100.0	0.2	0.7	2.8
20	100.0	5.3	35.2	17.7
30	99.6	84.4	567.0	48.4
40	94.8	217.6	589.9	73.4
50	80.0	419.9	585.2	39.5

Table 14.2: Computational results for $m = 1$

n	% solved	Avg. CPU	Max CPU	Avg. BCK
10	100.0	0.2	0.7	4.9
20	100.0	6.8	44.8	60.4
30	96.7	94.9	554.5	601.7
40	87.0	203.0	570.8	3252.5
50	74.4	475.8	597.0	947.0

Table 14.3: Computational results for $m = 3$

The algorithm behaves well since, on the average, 79.3 % of the 50 jobs instances can be solved in less than 10 minutes of CPU time. However, it appears that instances with parallel machines are much more difficult to solve than those with a single machine. This is due, we think, to the fact that our branching scheme is very efficient when we can use the algorithm of [54] for the feasibility check. A careful examination of the computational results shows that for $m > 1$ more than 90 % of the CPU time is spent in the feasibility check. Excellent results have been reported recently in [25] for the special case where $m = 1$ and $\forall i, w_i = 1$. Lower bounds based on Lagrangian relaxation and on relaxations of the problem to polynomially solvable cases are proposed. New elimination rules together with strong dominance relations are also used to reduce the search space. A branch and bound procedure exploiting all these techniques solves to optimality problems with up to 200 jobs.

14.2 Minimizing Total Tardiness

Consider the scheduling situation where n jobs $J_1, \ldots, J_n$ have to be processed by a single machine and where the objective is to minimize total tardiness. Associated with each job J_i, are a processing time p_i, a due date d_i, and a release date r_i. A job cannot start before its release date, preemption

n	% solved	Avg. CPU	Max CPU	Avg. BCK
10	100.0	0.1	0.2	0.2
20	100.0	9.9	67.0	238.3
30	98.2	70.5	551.6	1361.6
40	91.1	128.6	530.5	6676.6
50	84.8	336.4	539.4	6937.7

Table 14.4: Computational results for $m = 6$

is not allowed, and only one job at a time can be scheduled on the machine. The tardiness of a job J_i is defined as $T_i = \max(0, C_i - d_i)$, where C_i is the completion time of J_i. The problem is to find a feasible schedule with minimum total tardiness $\sum T_i$. The problem, denoted as $1|r_i|\sum T_i$, is known to be NP-hard in the strong sense.

A lot of research has been carried on the problem with equal release dates $1||\sum T_i$. Powerful dominance rules have been introduced by Emmons (see Section 11.2.2). Lawler [143] has proposed a dynamic programming algorithm that solves the problem in pseudo-polynomial time.

Most of the exact methods for solving $1||\sum T_i$ strongly rely on Emmons' dominance rules. Potts and Van Wassenhove [186], Chang *et al.* [73] and Szwarc *et al.* [208], have developed Branch-and-Bound methods using the Emmons rules coupled with the decomposition rule of Lawler [143] together with some other elimination rules. The best results have been obtained by Szwarc, Della Croce and Grosso [208] with a Branch-and-Bound method that efficiently handles instances with up to 500 jobs!

There are less results on the problem with arbitrary release dates. Chu and Portmann [74] have introduced a sufficient condition for local optimality which allows to build a dominant subset of schedules. Chu [75] has also proposed a Branch-and-Bound method using some efficient dominance rules. This method handles instances with up to 30 jobs for the hardest instances and with up to 230 jobs for the easiest ones.

In Section 14.2.1, we describe the overall branching scheme. In Sections 14.2.3 and 14.2.2, we introduce some dominance properties and we finally report our experimental results in Section 14.2.5. We will see that our procedure handles instances as large as 500 jobs although some 60 jobs instances remain open.

14.2.1 Overall Scheme

All propagation rules described in Section 11.2 are used. On top of this, we rely on the **edge-finding** branching scheme. Rather than searching for the starting times of jobs, we look for a **sequence** of jobs. This sequence is built both from the beginning and from the end of the schedule. Throughout the search tree, we dynamically maintain several sets of jobs that represent the current state of the schedule (see Figure 14.1).

- P is the sequence of the jobs scheduled at the beginning,
- Q is the sequence of the jobs scheduled at the end,
- NS is the set of unscheduled jobs that have to be sequenced between P and Q,
- $PF \subseteq NS$ (Possible First) is the set of jobs which can be scheduled immediately after P
- and $PL \subseteq NS$ (Possible Last) is the set of jobs which can be scheduled immediately before Q.

At each node of the search tree, a job J_i is chosen among those in PF and it is scheduled immediately after P. Upon backtracking, this job is removed from PF. The **heuristic** used to select J_i comes from [74, 75]: Among jobs of PF with minimal release date, select the job that minimizes the function $\max(r_i + p_i, d_i)$. Of course, if NS is empty then a solution has been found and we can iterate to the next decision problem. If $NS \neq \emptyset$ while PF or PL is empty then a backtrack occurs. Several propagation rules relying on jobs time-windows, are used to update and adjust these sets. These rules, known as edge-finding (see Chapter 8) are also able to adjust the time-windows according to the machine constraint.

Due to our branching scheme, jobs are sequenced from left to right so it may happen that at some node of the search tree, all jobs of NS have the same release date (the completion time of the last job in P). In such a case, to improve the behavior of the Branch-and-Bound method, we apply the dynamic programming algorithm of Lawler [143] to optimally complete the schedule.

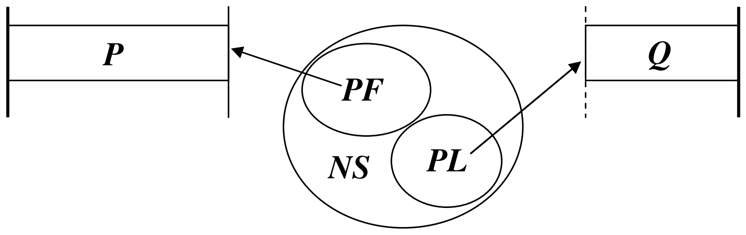


Figure 14.1: A Node of the Search Tree.

14.2.2 Dominance on Equal Length Jobs

We prove that, considering two jobs which have the same processing time, the first Emmons Rule (Section 11.2.2) is valid. Note that, contrary to the Generalized Emmons Rules, this dominance property is valid even in the non-preemptive case.

Proposition 92. *Let J_i and J_k be two jobs such that $p_i = p_k$. If $r_i \leq r_k$ and $d_i \leq \max(r_k + p_k, d_k)$, then there exists an optimal schedule in which J_i precedes J_k.*

Proof. Consider a schedule S in which job J_i and job J_k satisfy the assumptions and in which job J_k is completed before job J_i. Let us exchange J_i and J_k in S. Note that the exchange is valid since r_i is lower than or equal to the earliest start time of J_k. We have $p_i = p_k$, then the completion time of the other jobs do not change. This exchange does not increase total tardiness (see Section 11.2.2). □

14.2.3 Removing Dominated Sequences

Several dominance properties have been introduced in [74, 75]. These rules focus on the jobs in NS plus the last job of P. They determine that some precedence constraints can be added. Such constraints allow us to adjust release dates and to filter PF. All these rules are used in our Branch-and-Bound procedure. On top of this, we also consider dominance properties that take into account the **complete sequence** P. Informally speaking, our most basic rule states that if the current sequence P can be "improved", then it is dominated and we can backtrack.

In the following, let $C_{\max}(P)$ and $T(P)$ denote the completion time and the total tardiness associated with the current sequence P. Now consider a permutation P' of P of and let us examine under which condition P' is "as good as" than P.

- If $C_{\max}(P') \leq C_{\max}(P)$ and $T(P') \leq T(P)$, then we can replace P by P' in any feasible schedule so P' is at least as good as P.

- If $C_{\max}(P') > C_{\max}(P)$, then if we replace P by P' in a feasible schedule, all jobs in $NS \cup Q$ have to be shifted of at most $C_{\max}(P') - C_{\max}(P)$ time units. So, the additional cost for jobs in $NS \cup Q$ is at most $(|NS| + |Q|)(C_{\max}(P') - C_{\max}(P))$. Hence, P' is at least as good as P if $T(P') + (|NS| + |Q|)(C_{\max}(P') - C_{\max}(P)) \leq T(P)$.

P' is better than P if P' is at least as good as P and if either (1) P is not at least as good as P' (2) or if P' is lexicographically smaller than P. When P' is better than P, the current sequence is dominated and we can backtrack. To compare two sequences, we just have to build the schedules associated with them and this can be done in linear time.

Enumeration of the k Last Jobs

We can enumerate all permutations P' of P and test if it is better than P. Since the number of alternative sequences is exponential, we only consider the permutations P' that are identical to P except for the k last jobs (where k is an arbitrary value). When k is large, we have a great reduction of the search space but this takes a lot of time. Experimentally, $k = 6$ seems to be a good trade-off between the reduction of the search tree and the time spent in the enumeration.

Removing Possible Firsts

We propose a simple technique to detect that a job $J_i \in PF$ cannot be actually scheduled just after P and thus can be removed from PF. We note $P|J_i$ the sequence where J_i is scheduled immediately after P. If there is a permutation π of $P|J_i$ that is "better" than $P|J_i$ then scheduling J_i first after P leads to a dominated schedule and thus we can remove J_i from PF. This time we only enumerate the permutations π that are obtained from $P|J_i$ either by inserting J_i somewhere inside P or by exchanging J_i with another job of $P|J_i$. Since there are $O(n)$ such permutations and since comparing two sequences can be done in linear time, the algorithm runs in $O(n^2)$ for a given job J_i.

Adjusting Deadlines

Consider a job $J_i \in NS$ and assume that the current node of the search tree can be extended to a feasible optimal schedule where J_i is completed at some time point C_i. Let $P|\lambda|J_i|\mu|Q$ be the sequence of the jobs in the feasible optimal schedule and let us modify this sequence by removing J_i and inserting it somewhere in P. The new sequence is $P'|J_i|P''|\lambda|\mu|Q$, where $P'|J_i|P''$ is the sequence derived from P after the insertion of J_i.

In the following, we assume that $r_i \leq C_{\max}(P')$. Under this hypothesis, it is easy to see that $C_{\max}(P'|J_i|P''|\lambda) \leq C_{\max}(P|\lambda|J_i)$ and thus the **tardiness of the jobs in $\boldsymbol{\mu|Q}$** has not been increased. On top of that, the jobs in λ have been shifted of at most $C_{\max}(P'|J_i|P'') - C_{\max}(P)$ time units. Thus, the **total tardiness of the jobs in $\boldsymbol{\lambda}$** has been increased of at most $|\lambda|(C_{\max}(P'|J_i|P'') - C_{\max}(P)) \leq (|NS| - 1)(C_{\max}(P'|J_i|P'') - C_{\max}(P))$. Finally, the **total tardiness of the jobs in $\boldsymbol{P \cup \{J_i\}}$** has been increased of $T(P'|J_i|P'') - (T(P) + \max(0, C_i - d_i))$. Consequently, the total tardiness has been increased of at most

$$(|NS| - 1)(C_{\max}(P'|J_i|P'') - C_{\max}(P)) + T(P'|J_i|P'') - T(P) - \max(0, C_i - d_i).$$

Since the initial schedule is optimal, the above expression is positive and thus,

$$C_i < d_i + (|NS| - 1)(C_{\max}(P'|J_i|P'') - C_{\max}(P)) + T(P'|J_i|P'') - T(P).$$

In other words, the deadline $\bar{d}_i$ of J_i can be adjusted to

$$\min(\bar{d}_i, d_i - 1 + (|NS| - 1)(C_{\max}(P'|J_i|P'') - C_{\max}(P)) + T(P'|J_i|P'') - T(P)).$$

Since there are $O(n)$ sequences $P'|J_i|P''$ (with $r_i \leq C_{\max}(P')$) that can be derived from P and since the completion time and the total tardiness of a sequence can be computed in linear time, all adjustments of related to J_i can be computed in $O(n^2)$.

14.2.4 Intelligent Backtracking

Intelligent Backtracking techniques "record", throughout the search tree, some situations that do not lead to a feasible solution. Each time such a situation is encountered again then a backtrack is immediately triggered.

In practice, when a backtrack occurs, the set of "scheduled" jobs, *i.e.*, those in P, the tardiness associated to the sequence ($T(P)$) and and its completion time ($C_{\max}(P)$) are stored. Using a hash table for storing the set of schedule jobs, the total memory requirement is kept relatively low. If latter in the search tree, we are in a situation where a sequence P' of the same jobs as in P jobs has been built and if $T(P') \geq T(P)$ and $C_{\max}(P') \geq C_{\max}(P)$ then the current node of the search tree cannot lead to an optimal solution (otherwise, P could also be extended to an optimal sequence).

14.2.5 Experimental Results

All techniques presented below have been incorporated into a Branch-and-Bound method implemented on top of ILOG SOLVER and ILOG SCHEDULER. All experimental results have been computed on a PC Dell Latitude 650 MHz running Windows 98.

The instances have been generated with the scheme of Chu [75]. Each instance is generated randomly from three uniform distributions of r_i, p_i and d_i. The distribution of p_i is always between 1 and 10. The distributions of r_i and d_i depend on 2 parameters: α and β. For each job J_i, r_i is generated from the distribution $[0, \alpha \sum p_i]$ and $d_i - (r_i + p_i)$ is generated from the distribution $[0, \beta \sum p_i]$. Four values for α and three values for β were combined to produce 12 instances sets, each containing 10 instances of n jobs with $n \in \{10, 20, 30, 40, 50, \dots, 500\}$ jobs.

	Chu's Lower Bound				lb_2			
Opt.	lb_{Chu}	Gap	Bck.	CPU	lb_2	Gap	Bck.	CPU
34	28	0.21	501340	773.9	34	0.00	1	0.1
111	46	1.41	***	***	111	0.00	0	0.1
115	35	2.29	***	***	93	0.24	535	2.1
95	47	1.02	603171	1313.3	81	0.17	10300	59.1
32	19	0.68	24913	56.6	31	0.03	221	1.0
12	9	0.33	1	0.1	12	0.00	0	0.0
79	40	0.97	***	***	73	0.08	284	1.6
192	167	0.15	10118	28.2	180	0.07	462	2.4
84	54	0.56	52922	79.8	79	0.06	6	0.1
24	16	0.50	69754	89.4	24	0.00	2	0.1

Table 14.5: Comparison Between lb_{Chu} and lb_2. $n = 20$ Jobs, $\alpha = 0.5$, $\beta = 0.5$, no Dominance Property, no Intelligent Backtracking.

In Table 14.5, Chu's lower bound is compared with the new lower bound lb_2 (see Section 11.2.1). Most times, $lb_1 = lb_{Chu}$ so, lb_1 has been skipped from the tables to simplify the presentation. To have a fair comparison of the bounds, the Branch-and-Bound procedure has been run without dominance property nor Intelligent Backtracking on instances with 20 jobs that are known to be hard ($\alpha = 0.5, \beta = 0.5$) [75]. Each line corresponds to a single instance and we report the optimal tardiness (Opt.), the value of the lower bound computed at the root of search tree (lb_{Chu}, lb_2), the relative gap (Gap) between the optimum and the two lower bounds, the number of backtracks (Bck.) and the amount of CPU time in seconds required to solve the instance. A "***" in the table indicates that the search was interrupted after 1800 seconds. On the average, lb_{Chu} is at 81% of the optimum value while our lower bound is much closer (6%). The average number of backtracks over the instances solved by both methods has been divided by almost 12 and the CPU time by 37.

In Table 14.6, we show the efficiency of dominance properties, propagation rules and intelligent backtracking technique presented above. For that, the Branch-and-Bound procedure has been run with lb_2 on instances with 30 jobs with various combinations of α and β. On each line of Table 14.6, the average results obtained over the 10 generated instances are reported. In columns 3 and 4 ("Chu"), we report the results obtained when the dominance properties of Chu are used [74, 75]. We then add (columns 5 and 6) the dominance and propagation rules presented in Section 14.2.2 (EQual Processing, EQP), Section 14.2.3 (Remove Dominated Sequences, RDS), and Section 14.2.3 (optimization on the 6 last jobs). Intelligent Backtracking (IB) is then added and results are reported in columns 7 and 8. All "ingredients" are useful to reduce the search space.

The results obtained with the version of the algorithm that incorporates all ingredients are presented in Table 14.7. For each combination of parameters and for each value of n, we provide the average number of fails and the average computation time in seconds. A time limit of 3600 seconds has been fixed. All instances are solved within the time limit for up to 50 jobs. For $n = 60$, and for ($\alpha = 0.5, \beta = 0.5$), most of the instances cannot be solved. As noticed earlier by Chu [75], instances generated according to this particular combination seem to be "hard" to solve in practice.

		Chu		Chu, EQP, RDS 6-last, PIP		Chu, EQP, RDS 6-last, PIP, IB	
α	β	Bck.	CPU	Bck.	CPU	Bck.	CPU
0	0.05	0	0.1	0	0.0	0	0.0
0	0.25	7	0.1	7	0.1	7	0.1
0	0.5	80	0.6	80	0.6	80	0.6
0.5	0.05	91	0.7	26	0.3	24	0.3
0.5	0.25	5420	32.8	215	3.2	156	2.5
0.5	0.5	11505	70.1	424	6.2	296	4.6
1	0.05	239	1.4	31	0.3	24	0.3
1	0.25	1724	7.1	46	0.5	39	0.5
1	0.5	2	0.0	2	0.0	2	0.0
1.5	0.05	24	0.2	7	0.1	6	0.1
1.5	0.25	560	2.3	11	0.2	8	0.2
1.5	0.5	2	0.1	1	0.1	1	0.0

Table 14.6: Efficiency of Dominance Properties, Propagation Rules and Intelligent Backtracking.

From this table, we can remark that the "hardness" increases very quickly with n, especially for ($\alpha = 0.5$, $\beta = 0.5$). For Each combination of parameters, we report the largest size of instance (column "Largest") for which 80% of instances are solved within one hour of CPU time. In practice most of the instances are solved within 30 seconds and our results compare well to those of [75]. For instance, the average number of fails for the combination ($\alpha = 0.5, \beta = 0.5$) was greater than 36000 in [75], whereas this number is now lower than 300.

14.3 Minimizing Makespan and Sum of Transition Times

In this section we consider a General-Shop Problem with sequence-dependent setup times and alternative machines. The optimization criteria are both makespan and sum of setup times. The solution method and the computational study presented here are mostly taken from [103]. The solution method is based on a two phase algorithm where during the first phase we try to find a solution having as good a makespan as possible, and in the second phase the attention is turned to improving the sum of setup times while at least maintaining the quality of the makespan found in the first phase. It is shown that the constraint propagation as described in Section 11.3 considerably improves the performance of the approach.

14.3.1 Problem Definition

We are given a set of n jobs $\{J_1, \dots, J_n\}$ and a set of m machines $\{M_1, \dots, M_m\}$. Each job J_i has to be processed for p_i time units on a machine M_u that can be chosen from a given subset of the m machines. Between jobs, sequence-dependent setup times exist that also depend on which machine is used to process the jobs. Jobs may furthermore be linked by precedence relations $J_i \rightarrow J_j$, denoting that job J_j cannot start before the end of job J_i.

This problem obviously fits our Constraint-Based Scheduling model (see Section 7.2). Jobs are simply modeled by activities (to stay in the terminology of the problem we will continue to talk about jobs instead of activities in this section) and we have alternative machines for the jobs. The setup time between jobs J_i and J_j on machine M_u is represented by $\mathit{setup}(J_i, J_j, M_u)$, the setup time before J_i by $\mathit{setup}(-, J_i, M_u)$, and the teardown time after J_i by $\mathit{setup}(J_i, -, M_u)$ (see Section 7.2.4 for details).

The objective is first to find a schedule with minimal makespan, after which the sum of setup times is to be minimized. The constraint propagation on the sum of setup times that is used is

		$n = 10$		$n = 20$		$n = 30$		$n = 40$		$n = 50$		$n = 60$		Largest
α	β	Bck.	CPU	Bck.	CPU	Bck.	CPU	Bck.	CPU	Bck.	CPU	Bck.	CPU	
0	0.05	0.0	0.01	0	0.02	0	0.0	1	0.1	1	0.2	0	0.1	300
0	0.25	0.3	0.01	3	0.03	7	0.1	22	0.5	17	1.2	55	3.2	120
0	0.5	1.1	0.01	7	0.07	80	0.6	57	1.7	304	7.7	1591	91.3	80
0.5	0.05	2.9	0.02	16	0.08	24	0.3	73	1.8	96	3.9	238	19.7	90
0.5	0.25	3.6	0.02	26	0.18	156	2.5	484	21.4	1530	175.3	5253	1083.0	60
0.5	0.5	3.7	0.01	43	0.30	296	4.6	2366	131.7	9438	931.0	***	***	50
1	0.05	1.3	0.02	13	0.07	24	0.3	52	1.4	93	4.5	128	25.7	90
1	0.25	2.0	0.02	14	0.08	39	0.5	86	2.2	90	4.4	237	28.1	500
1	0.5	0.6	0.01	22	0.15	2	0.0	11	0.2	14	0.3	5	0.5	500
1.5	0.05	1.3	0.02	5	0.02	6	0.1	30	1.1	37	2.2	40	6.7	190
1.5	0.25	0.1	0.02	2	0.02	8	0.2	2	0.1	37	0.8	6	1.6	500
1.5	0.5	0.0	0.02	0	0.01	1	0.0	0	0.1	0	0.2	0	0.5	500

Table 14.7: Results Obtained with up to 60 Jobs.

described in Section 11.3.

14.3.2 Problem Solving

The problem is solved in two phases. During the first phase a good but not necessarily optimal solution with respect to makespan is searched for. With C^* the best makespan found in this phase, in the second phase a constraint is added imposing that any further solution will have a makespan smaller or equal to C^*. Constraint-based local improvement methods are then used to minimize the sum of setup times.

First Phase Heuristic

We use a time-limited, incomplete branch and bound method to try to find a solution that minimizes the makespan. At each node of the search tree we administer for each machine which job has been scheduled last. The set B contains all these jobs. By analyzing the precedence graph, we choose a job J_i among the set of jobs that are still unscheduled and can be next to one of the jobs in B. We branch on the relative position of J_i forcing J_i to be either **next** or **successor but not next** of one job in B. Among all jobs that can be chosen we select the one having the smallest earliest start time and, in case of ties, the one having the smallest latest end time. In the branch of the tree imposing J_i next to one job in B, we also need to choose the machine assignment for J_i. If several machine assignments are feasible we heuristically choose the machine assignment that allows to schedule the job as early as possible, and in case of ties, the one which generates the smallest setup time.

Setup Optimization

In the setup optimization phase, given a solution having a makespan equal to C^*, we search for solutions that have a makespan less or equal to C^* and that minimize the sum of setup times. For this the following local improvement procedure based on time windows is used. A time window $[TW_L^k, TW_U^k]$ defines a subproblem P_{TW^k} in the following way. On every machine, all jobs on the left of the window have their start times and machine assignments fixed; all jobs on the right of the time window have their machine assignment and their sequence on the machines fixed (the variable *next* is fixed); all jobs within the current window are completely free. On each subproblem a time-limited branch and bound search is performed to try to find the optimal solution for P_{TW^k}.

Two different methods have been used to select the current window. The first one is a simple "gliding window" method. The second one (LB-based Window Selection) relies on the computation of a lower bound of the sum of setup times to select the most promising window.

Gliding Window
Given a fixed size of the window W_{size} and a window offset W_d, the Gliding Window method starts the setup optimization at P_{TW^0} defined by window $[0, W_{size}]$, optimizes the problem, and then moves the window by W_d to the right, *i.e.*,

$$P_{TW^{k+1}} := [TW_L^k + W_d, TW_U^k + W_d]$$

This is repeated until the end of the schedule is reached. At the end of each loop, the window size and offset can eventually be modified and another loop can be performed.

LB-based Window Selection
This method is based on the idea to first work on the parts of the schedule where we can hope to obtain the highest improvement. For a given subproblem P_{TW^k}, defined by window $[TW_L^k, TW_U^k]$ we can calculate the expected improvement on the objective function E_{TW^k} as the difference between the current sum of setup times in that window, and the lower bound calculated in that window. After subproblem P_{TW^k} is defined, variable Z identifying the sum of setup times contains the information of the lower bound calculated by the constraint propagation. Crucial components of the constraint propagation are the route optimization constraint and the precedence graph constraint of

Section 11.3. So, if s^* is the total setup value of the current best solution found, E_{TW^k} is equal to $s^* - lb(Z)$. In the LB-based Window Selection, first E_{TW^k} is calculated for each subproblem P_{TW^k} as defined in the Gliding Window procedure. Then the subproblems are sorted in descending order of E_{TW^k}. All subproblems that may lead to an improvement of the objective function are labeled as improvable. We run the branch and bound algorithm on the first ranked subproblem that can lead to an improvement, and change the label of the window. If a better solution is found, the current solution is updated, and the values E_{TW^k} of all windows on the right of the modified one are recalculated since they may have been changed by the new solution. Also, the labels of the windows on the right are updated. The windows are then re-sorted and the procedure is repeated until no window exists that is labeled to be improvable.

14.3.3 Computational Results

The computational results in [103] that we present here show that the constraint propagation as described in Section 11.3 improves performance of the approach both in terms of computation time and quality of solutions. Other constraint propagation used for this problems includes the basic propagation on temporal relations and alternative machines as described in Section 7.2, the disjunctive constraint propagation of Section 8.1.2, and Edge-Finding as described in Section 8.1.3. It is shown that the large neighborhood defined by a time window containing between 30 and 60 jobs can be very efficiently explored thanks to this constraint propagation.

Instance generation

Following the computational studies in [51], experiments are run on Open-Shop, General-Shop, and Job-Shop Problems. The instances of [51] are used to test the approach on problems with setup times but without machine alternatives. These instances were duplicated and triplicated in [103] to generate scheduling problems with setup times and alternative machines. In order to generate an instance with the alternative choice of k machines, for each job and each machine in the original instance, k jobs and k machines are created. If in the original instance job J_i requires machine M_j, in the k–multiplied instance each one of the k identical jobs J_{ih} requires one out of the k identical machines M_{jh}. The temporal constraints among jobs are also duplicated, so that if a temporal constraint $J_i \to J_j$ exists in the original instance, the set of temporal constraints $J_{ih} \to J_{jh}$ exist in the new instance. For all instances without alternative machines the results of the approach described here can be compared to the results published in [51]. Nevertheless, a real comparison cannot be done since in [51] the objective is the minimization of the makespan, while here the aim is to minimize the sum of setup times in a problem constrained by a maximal makespan.

The Open-Shop instances with no alternative machines contain 8 machines and 8 jobs (16 machine and 16 jobs for the 2-alternative instances *etc.*). The General-Shop instances with no alternative machines also contain 8 machines and 8 jobs, and derive from the Open-Shop instances with the addition of temporal constraints (see [51]). The Job-Shop instances with no alternative machines contain 5 machines and 20 jobs.

Results

Tables 14.8, …, 14.16 report results on Open-Shop, General-Shop, and Job-Shop instances. Tables 14.8, 14.9, and 14.10 report results for the instances in [51]. The following tables report results for the duplicated and triplicated instances generated as described above. For each instance the results obtained by the first solution phase and the setup optimization phase are reported in terms of sum of setup times ($\sum tt$) and makespan (C^*). We also report in column B&T the results published in [51] in terms of makespan for all the instances with no alternative machine. The numbers in the tables that are in boldface indicate the best value found for the instance at hand. All tests were run on a Pentium II 300 MHz. The results published in [51] were obtained on a Sun 4/20 workstation, with a time limit of 5 hours for Open-Shop and General-Shop instances, and a time limit of 2 hours for Job-Shop instances.

Column "FirstSol" reports results in terms of makespan and sum of setup times of the best solutions obtained by the first solution phase. The time limit given was 60 seconds, and limited discrepancy search was used (see [116] and Section 12.3.4). The solution obtained after this first phase turns out to be a good solution with respect to makespan minimization. In half of the instances considered, the makespan found in the first solution phase improves the best known makespan published in [51].

As explained, the results of the first solution phase were used as a starting point for the setup optimization. In the setup optimization phase we fixed an initial window size of 30 jobs (*i.e.*, each subproblem has 30 completely free jobs), and we used a time limit of 5 seconds for the branch and bound algorithm to minimize the sum of setup times in each subproblem. In order to compare the results obtained with and without the constraint propagation described in Section 11.3, we used the same, very simple, branching strategy: we choose the variable *next* with the smallest domain and we branch on its possible values starting from the one generating the smallest setup time. Given an initial window size, the setup optimization methods (columns "GW", "GW+", "LBWS") are called until a local minimum is reached, then the window size is increased 20% (*e.g.*, from 30 free jobs to 36 free jobs), and the procedures are repeated until a global time limit is reached.

	FirstSol		GW		GW+		LBWS		B&T
	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*	C^*
TAIBS81	2680	942	1740	928	1620	919	**1480**	936	**914**
TAIBS85	3480	1113	2180	985	**1280**	1108	**1280**	1108	**899**
TAIS81	1460	699	980	693	**890**	**690**	980	693	713
TAIS85	1850	755	1260	748	**790**	748	850	754	**747**

Table 14.8: Open-Shop instances from [51]

	FirstSol		GW		GW+		LBWS		B&T
	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*	C^*
TAIBGS81	1680	763	**1410**	763	**1410**	763	1470	**759**	837
TAIBGS85	2010	869	1150	862	**870**	867	**870**	867	**762**
TAIGS81	1510	**734**	1190	**734**	**1150**	**734**	1190	**734**	858
TAIGS85	1540	749	1210	**745**	**1010**	747	1160	747	783

Table 14.9: General-Shop instances from [51]

	FirstSol		GW		GW+		LBWS		B&T
	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*	C^*
T2-PS12	1710	1450	1640	1450	1640	1450	**1530**	**1448**	1528
T2-PS13	1930	1669	1640	1667	1640	1667	**1430**	1658	**1549**
T2-PSS12	1480	1367	1300	1367	1300	1367	**1220**	**1362**	1384
T2-PSS12	1290	1522	1220	1522	**1140**	1522	1220	1518	**1463**

Table 14.10: Job-Shop instances from [51]

Column "GW" and column "GW+" report the results obtained by the Gliding Window method described in Section 14.3.2. The algorithm used for column "GW" does not use the constraint propagation of Section 11.3, while the algorithm used for column "GW+" does. Column "LBWS" reports the results obtained by the LB-based Window Selection method (also described in Section 14.3.2). For the Open-Shop and General-Shop instances of Tables 14.8 and 14.9 (containing 64 jobs each), the global time limit used is 30 seconds. For the Job-Shop instances of Table 14.10 (containing 100 jobs each), and for the Open-Shop and General-Shop instances of Tables 14.11 and 14.12 (containing 128 jobs each), the global time limit used is 60 seconds. For the Job-Shop instances of Table 14.13 (containing 200 jobs each), and for the Open-Shop and General-Shop instances of Tables 14.14 and 14.15 (containing 192 jobs each), the global time limit used was 120 seconds. For the Job-Shop instances of Table 14.16 (containing 300 jobs each), the global time limit used is 240 seconds.

When the propagation algorithms of Section 11.3 are used ("GW+"), the solutions obtained

	FirstSol		GW		GW+		LBWS	
	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*
TAIBS81	3920	908	3760	903	**2760**	904	2840	905
TAIBS85	4520	942	4260	940	2580	939	**2380**	942
TAIS81	2220	723	2060	723	**1540**	723	1590	723
TAIS85	2280	690	2110	689	**1730**	690	1950	689

Table 14.11: Open-Shop instances with two alternative machines

	FirstSol		GW		GW+		LBWS	
	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*
TAIBGS81	2220	1023	2140	1023	**1270**	1008	1490	1017
TAIBGS85	2640	1031	2350	1019	1300	1020	**1150**	1026
TAIGS81	2510	766	2430	764	1900	766	**1720**	756
TAIGS85	2490	748	2450	748	1810	743	**1710**	748

Table 14.12: General-Shop instances with two alternative machines

	FirstSol		GW		GW+		LBWS	
	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*
T2-PS12	3410	1562	2980	1537	**2330**	1552	2510	1551
T2-PS13	2890	1593	2670	1593	2270	1584	**2240**	1593
T2-PSS12	2090	1515	1820	1479	1610	1505	**1540**	1515
T2-PSS12	2120	1578	1720	1576	**1520**	1574	1590	1545

Table 14.13: Job-Shop instances with two alternative machines

	FirstSol		GW		GW+		LBWS	
	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*
TAIBS81	4780	1002	4380	999	**3320**	999	3520	986
TAIBS85	5320	875	5280	875	4180	870	**4160**	865
TAIS81	2910	802	2440	802	2190	800	**2090**	802
TAIS85	2660	758	2540	758	2020	755	**1690**	757

Table 14.14: Open-Shop instances with three alternative machines

	FirstSol		GW		GW+		LBWS	
	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*
TAIBGS81	2230	1083	2140	1067	**1380**	1079	1540	1083
TAIBGS85	2240	1280	2080	1280	1550	1268	**1410**	1268
TAIGS81	2470	887	2430	887	1740	887	**1670**	885
TAIGS85	2900	789	2710	789	**1760**	789	1850	787

Table 14.15: General-Shop instances with three alternative machines

	FirstSol		GW		GW+		LBWS	
	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*	$\sum tt$	C^*
T2-PS12	2870	1593	2740	1593	2640	1587	**2210**	1585
T2-PS13	2600	1585	2600	1585	**2400**	1585	2500	1585
T2-PSS12	2500	1455	2360	1455	**2240**	1455	2290	1455
T2-PSS12	2100	1562	1850	1562	1770	1562	**1730**	1562

Table 14.16: Job-Shop instances with three alternative machines

consistently have at least the quality obtained when this propagation is not used ("GW"), and very often the quality is significantly better. The improvement in the solution quality is particularly important for problems with two and three alternative machines. Problems with no alternative machines are easier and even without the extra propagation, in each window the local optimal solution can often be found. However, even in these cases, when the extra constraint propagation is used the subproblems are solved up to optimality in a shorter time.

A comparison between the LBWS and GW+ shows that the best solutions are equally distributed between the two. Thus neither of the two methods clearly dominates the other. What one could remark is that if the scheduling problem is small enough to allow one or several complete gliding window loops, the LB-based method may lose some interest: if all windows are considered, the order in which they are solved may not be too important. On the other hand, for larger problems a complete gliding window loop may not be possible within the CPU time available. In such cases, a fast evaluation of the most promising area for improvement may play an important role. For more details on the performance improvements in these tests due to using the constraint propagation of Section 11.3, we refer to [103].

Although the aim of the methods proposed is to study the problem including alternative machines, some additional tests were run on small instances with no alternative machines taken from [51]. Open-Shop and General-Shop instances contain 4 machines and 4 jobs, while Job-Shop instances contain 5 machines and 10 jobs. For each instance two results are reported in Table 14.17. Column "C^* before $\sum tt$" reports the results obtained by first optimizing the makespan, and then given this makespan optimizing the sum of setup times. Column "$\sum tt$ before C^*" reports the results obtained by first optimizing the sum of setup times, and then given this sum of setup times optimizing the makespan.

	C^* before $\sum tt$			$\sum tt$ before C^*		
	$\sum tt$	C^*	time	$\sum tt$	C^*	time
Open-Shop						
TAIBS01	380	306	0.38	320	384	2.58
TAIBS05	440	395	0.22	320	563	0.72
TAIS01	190	249	0.28	160	344	1.93
TAIS05	220	348	0.27	160	523	0.33
General-Shop						
TAIBGS01	280	322	0.16	160	527	0.06
TAIBGS05	280	491	0.33	160	747	0.06
TAIGS01	230	285	0.44	160	362	0.39
TAIGS05	300	384	0.22	160	546	0.16
Job-Shop						
T2-PS01	710	798	-	250	2368	-
T2-PS02	630	784	88.42	250	2221	-
T2-PS03	550	749	144.20	250	1932	-
T2-PS04	670	730	388.31	250	1665	-
T2-PS05	710	691	30.43	250	1899	-

Table 14.17: Results on "small" instances

In all cases where a computation time is reported, the optimal solution could be proven for both criteria. For example, the optimal makespan of TAIBS01 is 306, and given such a makespan the optimal sum of setup times is 380; on the other hand, the optimal sum of setup times of TAIBS01 is 320, and given this sum of setup times the optimal makespan is 384. Where the time is not reported, optimality could not be proven within 30 minutes.

Two conclusions can be drawn from these results. First, the minimization of only one objective may generate poor quality solutions for the other objective. This indicates the usefulness of considering multi-criteria objectives in future research. Second, it appears that for small problems it is possible to fix any limit for one objective and find the optimal solution for the other objective.

Therefore the method described here might be usable to optimize any combination of makespan and sum of setup times and find a set of Pareto-optimal solutions.

Chapter 15

Conclusion

The various examples provided in Chapters 6, 7, and 8 have shown how a general Constraint-Based Scheduling model and the associated constraint propagation algorithms can be used to enable the resolution of complex scheduling problems.

On **disjunctive** scheduling problems (*e.g.*, Job-Shop, Open-Shop), the CP approach is competitive with classical OR techniques. Indeed, the most crucial "ingredients" from the successful OR algorithms (*e.g.*, Edge-Finding) have been adapted to the CP framework and are now routinely used as part of industrial constraint-based applications. The CP framework also enables the resolution of NP-hard **preemptive** problems (*e.g.*, Preemptive Job-Shop). We are not aware of any other practical attempt to solve such problems. In this latter case, the generalization of non-preemptive techniques to a less constrained situation has enabled the implementation of the first efficient problem-solving algorithm for Preemptive Job-Shop Scheduling.

On **cumulative** problems, computational studies have shown that good results can be obtained for most problem instances. Yet it appears that dedicated OR algorithms are still more efficient than CP on some problem instances (for example, on "highly disjunctive" instances of Section 13.3). Understanding the "structure" of the instances that resist to current CP tools and designing new techniques (*e.g.*, new lower-bounding and new constraint propagation algorithms) to handle these instances constitute a major research direction for the next few years.

Chapters 5 and 8 focused on the importance of **objective functions** and showed how they can be integrated in a constraint-based framework. Very good results have been obtained on some particular problems (*e.g.*, total weighted number of late jobs, sum of transition times) but of course many other objective functions (*e.g.*, total tardiness, total flow-time) still have to be studied. Designers of scheduling applications often have to define their own criteria. A research challenge is to design generic lower-bounding techniques and constraint propagation algorithms that could work for many criteria.

Similarly, designers of scheduling applications often have to define and add their own constraints to the basic Constraint-Based Scheduling model. The strength of the CP framework in this situation is that the results of propagating the additional constraints are automatically combined with the results of the deductive rules and algorithms presented in this manuscript, thereby creating a "snowball" effect: As long as one constraint can contribute to the reduction of the domain of one variable, the constraint propagation process continues and the domains are reduced. In this sense, additional constraints do not prevent the use of predefined constraint propagation algorithms (available in libraries) and contribute to making these algorithms more effective. In practice, three issues must however be taken into account:

- For complex additional constraints, the design and the implementation of efficient constraint propagation algorithms is not an easy task. In practice, if more than one scheduling application includes a given type of constraint, it might be worth developing **once** an efficient version of the corresponding constraint propagation algorithms. Examples of scheduling constraints that do not appear in the model of Chapter 1, but appear in libraries, include *energy resource constraints*, stating that the energy available from a resource over intervals of a given length is limited (*e.g.*, an employee works no more than 40 hours a week); *energy temporal constraints*, stating that at least some amount of energy must be spent by a given resource on a given activity before or after the beginning or the end of another activity; *state resource constraints*, enabling two activities to overlap only if they require a given resource in the same state (*e.g.*, two products can be cooked in the same oven at the same time only if they require the same oven temperature); *reservoir constraints*, stating that the level of a given reservoir, filled and unfilled by given activities, must remain over a given limit; *etc.*

- Constraint propagation, especially when complex algorithms are used, takes time. In general, the more complex constraint propagation algorithms prune more values from the domains of the variables but take more time to execute. When search is performed, this results in a smaller search tree but with more time spent at each node. The best combination of constraint propagation algorithms depends on the application and, for the same application, can vary from a problem instance to another. When several constraint propagation algorithms exist for the same constraint, the designer of the application must therefore decide which combination of algorithms is going to be used. This can usually be done by intuition (if the designer knows which constraints will be the most difficult to satisfy) and experimentation (assuming a representative set of problem instances is available). Theoretical "domination" results (*e.g.*, the results presented in Chapter 4) and experimental results from the literature can also be useful. Although necessarily incomplete, the theoretical and experimental comparisons provided in this manuscript hopefully provide a clear overview of the state of the art in the field and clarify a rather complex research situation.

- In some cases, it can be worth developing a global propagation algorithm for a collection of constraints. This is typically what is done in Chapter 5, where the resource constraint and the constraint defining the objective function are mixed to make constraint propagation more effective. Another interesting case is the combination of temporal and resource constraints (*cf.*, for example, [179, 204, 138]). As previously, if more than one scheduling application includes a given combination of constraints, it might be worth developing an efficient constraint propagation algorithm for this combination.

This manuscript dealt with combining OR and AI techniques in a way that preserves the best of both, *i.e.*, the efficiency provided by OR and the generality of approach offered by AI. Some readers might wonder why little has been said about various forms of local search, such as simulated annealing, tabu search, genetic algorithms, *etc.*, which provide excellent results when one can define a compact representation of the solution space that is consistent with the objective function. In real life, problems often incorporate side constraints that tend to disable the local search approach. This led several researchers to integrate CP and local search techniques (*cf.*, for example, [68, 71] in the scheduling domain and [183, 200, 128, 70, 71] in the vehicle routing domain for which this approach has been the most successful).

Similarly, little has been said about Linear Programming (LP) and Mixed Integer Programming (MIP). As a matter of fact, several constraint propagation algorithms in the manuscript do solve linear programs, but usually these programs have enough structure to allow the design of specific algorithms with low complexity. Obviously, this does not mean that LP is useless in the context of Constraint-Based Scheduling.

- For cumulative problems, the complexity of specific constraint propagation algorithms tends to raise (*e.g.*, to $O(n^3)$), which suggests that lower-bounding and constraint propagation algorithms based on LP might be competitive. However, most of the existing lower bounds based on LP are still very time consuming and cannot be computed at each node of a search tree.

- LP can also be a strong "ingredient" when the objective function is a sum or a weighted sum of scheduling variables like the end times of activities. A key research issue here is the design of techniques combining the power of efficient constraint propagation algorithms for the resource constraints and the power of LP for bounding the objective function.

- In real-life applications, scheduling issues are often mixed with resource allocation, capacity planning, or inventory management issues for which MIP is a method of choice. Several examples have been reported where a hybrid combination of CP and MIP was shown to be more efficient than pure CP or MIP models (*cf.*, for example, [195, 196, 94]). The generalization of these examples into a principled approach is another important research issue for the forthcoming years.

Chapter 16

Summary of notation

We summarize all notation used in Part 2 and some of those used in Part 1.

Symbol	Definition
r_i	Release date of job J_i
$\bar{d}_i$	Deadline of J_i
d_i	Due-date of J_i
p_i	Processing time of J_i
w_i	Weight associated to J_i
C_i	Completion time of J_i
T_i	Tardiness of J_i, $T_i = \max(0, C_i - \bar{d}_i)$
E_i	Earliness of J_i, $E_i = \max(0, \bar{d}_i - C_i)$
L_i	Lateness of J_i, $L_i = C_i - \bar{d}_i$

Table 16.1: Some standard scheduling notations

Symbol	Definition
A_i	Activity
n	Total number of activities
R	Resource

Table 16.2: Activities and resources

Symbol	Definition
criterion	Variable representing the criterion to be minimized
$cap(R)$, *cap*	Variable representing the capacity of the resource (R)
$cap(A_i, R)$, $cap(A_i)$	Variable representing the number of units of the resource (R) required by A_i throughout its execution
$E(A_i, R)$, $E(A_i)$	Variable representing the energy of the resource (R) required by A_i
$start(A_i)$	Variable representing the starting time of A_i
$end(A_i)$	Variable representing the ending time of A_i
$proc(A_i)$	Variable representing the processing time of A_i
$altern(A_i)$	Variable representing the alternative resource on which A_i will be executed (used only when there are alternative resources)
$setup(A_i, A_j)$	Amount of time that must elapse between the end of A_i and the start of A_j
$setupCost(A_i, A_j)$	Cost associated to the transition between A_i and A_j
$set(A_i)$	Variable representing the set of points at which A_i executes
$X(A_i, t)$	Binary variable that equals 1 if and only if A_i is in process at time t
$E(A_i, t, R)$, $E(A_i, t)$	Variable representing the number of units of the resource (R) used by activity A_i at time t

Table 16.3: Variables of the Constraint-Based Scheduling model

Symbol	Definition
C_R, C	Maximum value in the domain of $cap(R)$
$c_{i,R}, c_i$	Minimum value in the domain of $cap(A_i, R)$
$e_{i,R}, e_i$	Minimum value in the domain of $E(A_i, R)$
p_i	Minimum value in the domain of $proc(A_i)$
r_i	Release date (or earliest start time) of A_i
lst_i	Latest start time of A_i
eet_i	Earliest end time of A_i
$\bar{d}_i$	Deadline (or latest end time) of A_i

Table 16.4: Bounds of the variables used in the Constraint-Based Scheduling model

Symbol	Definition
p_Ω	Sum of the processing times p_i of activities in Ω
e_Ω	Sum of the energies e_i of activities in Ω
r_Ω	Smallest release date among those of activities of Ω
$eetmin_\Omega$	Smallest value in $\{eet_i : A_i \in \Omega\}$
$\bar{d}_\Omega$	Largest deadline among those of activities of Ω
$lstmax_\Omega$	Largest value in $\{lst_i : A_i \in \Omega\}$

Table 16.5: Values related to a set Ω of activities

Symbol	Definition
$domain(X)$	Domain of the variable X
$lb(X)$	Smallest value in the domain of the variable X
$ub(X)$	Largest value in the domain of the variable X
$\min(S)$	Minimal value in the set S
$\max(S)$	Maximal value in the set S
$length(I)$	Length of the interval I

Table 16.6: Basic notations

Symbol	Definition
$W_{PE}(A_i, t_1, t_2)$	Energy consumption of A_i over $[t_1, t_2)$ (preemptive case) $= c_i \max(0, p_i - \max(0, t_1 - r_i) - \max(0, \bar{d}_i - t_2))$
$W_{PE}(t_1, t_2)$	Overall required energy consumption over $[t_1, t_2)$ $= \sum_i W_{PE}(A_i, t_1, t_2)$
$S_{PE}(t_1, t_2)$	Slack over $[t_1, t_2)$ (preemptive case) $= C(t_2 - t_1) - W_{PE}(t_1, t_2)$
$p_i^+(t_1)$	Minimal number of time units during which A_i must execute after t_1 $= \max(0, p_i - \max(0, t_1 - r_i))$
$p_i^-(t_2)$	Minimal number of time units during which A_i must execute before t_2 $= \max(0, p_i - \max(0, \bar{d}_i - t_2))$
$W_{Sh}(A_i, t_1, t_2)$	Energy consumption of A_i over $[t_1, t_2)$ (non-preemptive case) $= c_i \min(t_2 - t_1, p_i^+(t_1), p_i^-(t_2))$
$W_{Sh}(t_1, t_2)$	Overall required energy consumption over $[t_1, t_2)$ $= \sum_i W_{Sh}(A_i, t_1, t_2)$
$S_{Sh}(t_1, t_2)$	Slack over $[t_1, t_2)$ (non-preemptive case) $= C(t_2 - t_1) - W_{Sh}(t_1, t_2)$

Table 16.7: Notations related to energetic reasoning

Bibliography

[1] E. H. L. Aarts and J. K. Lenstra. *Local Search in Combinatorial Optimization.* John Wiley and Sons, 1994.

[2] T.S. Abdul-Razacq, C.N. Potts and L.N. van Wassenhove. A survey of algorithms for the single machine total weighted tardiness scheduling problem. *Discrete Applied Mathematics*, 26:235–253, 1990.

[3] J. Adams, E. Balas and D. Zawack. The Shifting Bottleneck Procedure for Job-Shop Scheduling. *Management Science*, 34(3):391–401, 1988.

[4] A. Aggoun and N. Beldiceanu. Extending CHIP in Order to Solve Complex Scheduling and Placement Problems. *Mathematical and Computer Modelling*, 17(7):57–73, 1993.

[5] R. K. Ahuja, T. L. Magnanti and J. B. Orlin. *Network Flows.* Prentice Hall, 1993.

[6] S. Albers and P. Brucker. The Complexity of One-Machine Batching Problems. *Discrete Aplieded Mathematics* 47:87–107, 1993.

[7] A. Allahverdi, J. Gupta and T. Aldowaisan. A review of scheduling research involving setup consideration. *Omega*, 27(2):219-239, 1999.

[8] D. Applegate and W. Cook. A Computational Study of the Job-Shop Scheduling Problem. *ORSA Journal on Computing*, 3(2):149–156, 1991.

[9] K. R. Baker. *Introduction to Sequencing and Scheduling.* John Wiley and Sons, 1974.

[10] Ph. Baptiste. *Resource Constraints for Preemptive and Non-Preemptive Scheduling.* MSc Thesis, University of Paris VI, 1995.

[11] Ph. Baptiste. *A Theoretical and Experimental Study of Resource Constraint Propagation.* PhD Thesis, University of Technology of Compiègne, 1998.

[12] Ph. Baptiste. Polynomial Time Algorithms for Minimizing the Weighted Number of Late Jobs on a Single Machine when Processing Times are Equal. *Journal of Scheduling*, 2:245–252, 1999.

[13] Ph. Baptiste. Scheduling Equal-Length Jobs on Identical Parallel Machines. *Discrete Applied Mathematics*, 103:21–32, 2000.

[14] Ph. Baptiste. An $O(n^4)$ Algorithm for Preemptive Scheduling of a Single Machine to Minimize the Number of Late Jobs. *Operations Research Letters*, 24:175–180, 1999.

[15] Ph. Baptiste. Preemptive Scheduling of Identical Machines. *Research Report 314, University of Technology of Compiègne*, 2000.

[16] Ph. Baptiste, J. Carlier and A. Jouglet. A Branch-and-Bound Procedure to Minimize Total Tardiness on One Machine with Arbitrary Release Dates. Submitted to *European Journal of Operational Research*, 2002.

[17] Ph. Baptiste, A. Jouglet, C. Le Pape and W. Nuijten. A Constraint-Based Approach to Minimize the Weighted Number of Late Jobs on Parallel Machines. *Research Report 288, University of Technology of Compiègne*, 2000.

[18] Ph. Baptiste and A. Jouglet. On minimizing total tardiness in a serial batching problem. *RAIRO Operations Research* 35:107–115, 2001.

[19] Ph. Baptiste and C. Le Pape. A Theoretical and Experimental Comparison of Constraint Propagation Techniques for Disjunctive Scheduling. *Proc. 14th International Joint Conference on Artificial Intelligence*, 1995.

[20] Ph. Baptiste and C. Le Pape. Disjunctive Constraints for Manufacturing Scheduling: Principles and Extensions. *International Journal of Computer Integrated Manufacturing*, 9(4):306–310, 1996.

[21] Ph. Baptiste and C. Le Pape. Edge-Finding Constraint Propagation Algorithms for Disjunctive and Cumulative Scheduling. *Proc. 15th Workshop of the UK Planning Special Interest Group*, 1996.

[22] Ph. Baptiste and C. Le Pape. Constraint Propagation and Decomposition Techniques for Highly Disjunctive and Highly Cumulative Project Scheduling Problems. *Proc. 3rd. International Conference on Principles and Practice of Constraint Programming*, 1997.

[23] Ph. Baptiste, C. Le Pape and W. Nuijten. Satisfiability Tests and Time Bound Adjustments for Cumulative Scheduling Problems. *Annals of Operations Research*, 92:305–333, 1999.

[24] Ph. Baptiste, C. Le Pape and L. Péridy. Global Constraints for Partial CSPs: A Case Study of Resource and Due-Date Constraints. *Proc. 4th International Conference on Principles and Practice of Constraint Programming*, 1998.

[25] Ph. Baptiste, L. Peridy and E. Pinson. A Branch and Bound to Mininimze the Number of Late Jobs on a Single Machine with Release Time Constraints. *Research Report, University of Technology of Compiègne*, 2000.

[26] Ph. Baptiste and V. Timkovsky. On Preemption Redundancy in Scheduling Unit Processing Time Jobs on Two Parallel Machines. *Operations Research Letters*, 28:205–212, 2001.

[27] Ph. Baptiste and V. Timkovsky. Thirteen Notes on Equal–Execution–Time Scheduling. Submitted to *4OR*, 2002.

[28] A. Bar-Noy, S. Guha, Y. Katz, J. S. Naor B. Schieber and H. Shachnai. Throughput Maximization of Real-Time Scheduling with Batching. *ACM-SIAM Symposium on Discrete Algorithms*, 2002.

[29] H. Beck. Constraint Monitoring in TOSCA. *Working Papers of the AAAI Spring Symposium on Practical Approaches to Planning and Scheduling*, 1992.

[30] J. C. Beck, A. J. Davenport, E. M. Sitarski and M. S. Fox. Texture-Based Heuristics for Scheduling Revisited. *Proc. 14th National Conference on Artificial Intelligence*, 1997.

[31] N. Beldiceanu and E. Contejean. Introducing Global Constraints in CHIP. *Mathematical and Computer Modelling*, 20(12):97–123, 1994.

[32] R. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[33] C. Bessière, E. Freuder and J.-C. Régin. Using Inference to Reduce Arc Consistency Computation. *Proc. 14th International Joint Conference on Artificial Intelligence*, 1995.

[34] J. Blazewicz, W. Domschke and E. Pesch. The Job-Shop Scheduling Problem: Conventional and New Solution Techniques. *European Journal of Operational Research*, 93(1):1–33, 1996.

[35] J. Blazewicz, K. H. Ecker, E. Pesch, G. Schmidt and J. Weglarz. *Scheduling Computer and Manufacturing Processes.* 2nd ed., Springer, 2001.

[36] S. C. Brailsford, C. N. Potts and B. M. Smith. Constraint Satisfaction Problems: Algorithms and Applications. *European Journal of Operational Research*, 119:557-581, 1999.

[37] W. Brinkkötter and P. Brucker. Solving Open Benchmark Problems for the Job Shop Problem. *Journal of Scheduling*, 4:53–64, 2001.

[38] P. Brucker. *Scheduling Algorithms.* Springer Lehrbuch, 2001.

[39] P. Brucker, A. Gladky, H. Hoogeveen, M. Kovalyov, C. Potts, T. Tautenhahn and S. van de Velde. Scheduling a Batching Machine. *Journal of Scheduling* 1:31–54, 1998.

[40] P. Brucker, S. Heitmann and J. Hurink, How Useful are Preemptive Schedules? *Osnabrücker Schriften zur Mathematik*, Reihe P, No. 240 (2001).

[41] P. Brucker, J. Hurink, B. Jurisch and B. Wostmann. A branch and bound algorithm for the open shop problem. *Discrete Applied Mathematics*, 76:43–59, 1997.

[42] P. Brucker, J. Hurink and S. Knust. A polynomial algorithm for $P|p_j = 1, r_j, outtree|\sum C_j$. *Technical Report, University of Osnabrueck*, Germany, 2001.

[43] P. Brucker, B. Jurisch, M. Jurisch. Open Shop Problems With Unit Time Operations. *Zeitschrift für Operations Research* 37:59–73, 1993.

[44] P. Brucker, B. Jurisch and B. Sievers. A Branch & Bound Algorithm for the Job-Shop Scheduling Problem. *Discrete Applied Mathematics*, 49:107–127, 1992.

[45] P. Brucker, B. Jurisch, T. Tautenhahn and F. Werner. Scheduling unit time open shops to minimize the weighted number of late jobs. *Operations Research Letters* 14:245–250, 1993.

[46] P. Brucker and S. Knust. A Linear Programming and Constraint Propagation-Based Lower Bound for the RCPSP. *European Journal of Operational Research*, 127:355–362, 2000.

[47] P. Brucker and S. Knust, Complexity Results of Scheduling Problems, mathematik.uni-osnabrueck.de/research/OR/class.

[48] P. Brucker, S. Knust, D. Roper and Y. Zinder. Scheduling UET task systems with concurrency on two parallel identical processors. *Mathematical Methods of Operations Research* 53:369–387, 2000.

[49] P. Brucker and M.Y. Kovalyov. Single machine batch scheduling to minimize the weighted number of late jobs. *Mathematical Methods of Operations Research* 43:1–8, 1996.

[50] P. Brucker and A. Krämer. Polynomial algorithms for resource-constrained and multiprocessor task scheduling problems. *European Journal of Operational Research*, 90:214–226, 1996.

[51] P. Brucker and O. Thiele. A Branch and Bound Method for the General-Shop Problem with Sequence-Dependent Setup Times. *OR Spektrum*, 18:145–161, 1996.

[52] M. Bruynooghe. Solving Combinatorial Search Problems by Intelligent Backtracking. *Information Processing Letters*, 12(1):36–39, 1981.

[53] J. Carlier. Problème à une machine et algorithmes polynômiaux. *QUESTIO*, 5(4):219–228, 1981.

[54] J. Carlier. The One-Machine Sequencing Problem. *European Journal of Operational Research*, 11:42–47, 1982.

[55] J. Carlier. *Problèmes d'Ordonnancement à Contraintes de Ressources : Algorithmes et Complexité.* Thèse de Doctorat d'Etat, Université Paris VI, 1984.

[56] J. Carlier and Ph. Chrétienne. *Problèmes d'ordonnancement : Modélisation / Complexité / Algorithmes.* Masson, 1988.

[57] J. Carlier and B. Latapie. Une méthode arborescente pour résoudre les problèmes cumulatifs. *RAIRO Recherche Opérationnelle*, 25(3):311–340, 1991.

[58] J. Carlier and E. Néron. A New Branch and Bound Method for Solving the Resource-Constrained Project Scheduling Problem. *Proc. International Workshop on Production Planning and Control*, 1996.

[59] J. Carlier and E. Néron. An Exact Method for Solving the Multi-Processor Flow-Shop. *RAIRO Recherche Opérationnelle*, 34:1–25, 2000.

[60] J. Carlier and E. Néron. A New LP Based Lower Bound for the Cumulative Scheduling Problem. *Research Report, University of Technology of Compiègne*, 2001.

[61] J. Carlier and E. Pinson. An Algorithm for Solving the Job-Shop Problem. *Management Science*, 35(2):164–176, 1989.

[62] J. Carlier and E. Pinson. A Practical Use of Jackson's Preemptive Schedule for Solving the Job-Shop Problem. *Annals of Operations Research*, 26:269–287, 1990.

[63] J. Carlier and E. Pinson. Adjustment of Heads and Tails for the Job-Shop Problem. *European Journal of Operational Research*, 78:146–161, 1994.

[64] J. Carlier and E. Pinson. Jackson's Pseudo-Preemptive Schedule for the $Pm|r_i, q_i|C_{\max}$ Scheduling Problem. *Annals of Operations Research*, 83:41–58, 1998.

[65] G. Carpaneto, S. Martello and P. Toth. Algorithms and code for the assignment problem. *Annals of Operations Research*, 13:193–223, 1988.

[66] Y. Caseau. Contraintes et algorithmes, petit précis d'optimisation combinatoire pratique. *Notes de cours du Magistère de Mathématiques Fondamentales et Appliquées et d'Informatique, Ecole Normale Supérieure*, 1996.

[67] Y. Caseau and F. Laburthe. Improved CLP Scheduling with Task Intervals. *Proc. 11th International Conference on Logic Programming*, 1994.

[68] Y. Caseau and F. Laburthe. Disjunctive Scheduling with Task Intervals. *Technical Report, Ecole Normale Supérieure*, 1995.

[69] Y. Caseau and F. Laburthe. CLAIRE: A Parametric Tool to Generate C++ Code for Problem Solving. *Working Paper, Bouygues, Direction Scientifique*, 1996.

[70] Y. Caseau and F. Laburthe. Heuristics for Large Constrained Vehicle Routing Problems. *Journal of Heuristics*, 5:281-303, 1999.

[71] Y. Caseau, F. Laburthe, C. Le Pape and B. Rottembourg. Combining Local and Global Search in a Constraint Programming Environment. *Knowledge Engineering Review*, to appear.

[72] A. Cesta and A. Oddi. Gaining Efficiency and Flexibility in the Simple Temporal Problem. *Proc. 3rd International Workshop on Temporal Representation and Reasoning*, 1996.

[73] S. Chang, Q. Lu, G. Tang and W. Yu. On decomposition of the total tardiness problem. *Operations Research*, 17:221–229., 1995.

[74] C. Chu and M.-C. Portmann. Some new efficient methods to solve the $n|1|r_i|\sum T_i$ scheduling problem. *European Journal of Operational Research* 58:404–413, 1991.

[75] C. Chu. A Branch-and-Bound algorithm to minimize total tardiness with different release dates. *Naval Research Logistics*, 39:265–283, 1992.

[76] E.G. Coffman, M. Yannakakis, M.J. Magazine and C. Santos. Batch sizing and sequencing on a single machine. *Annals of Operations Research* 26:135–147, 1990.

[77] A. Colmerauer. An Introduction to PROLOG III. *Communications of the ACM*, 33(7):69–90, 1990.

[78] Y. Colombani. Constraint Programming: An Efficient and Practical Approach to Solving the Job-Shop Problem. *Proc. 2nd International Conference on Principles and Practice of Constraint Programming*, 1996.

[79] S. Dauzère-Pérès. Minimizing Late Jobs in the General One-Machine Scheduling Problem. *European Journal of Operational Research*, 81:134–142, 1995.

[80] S. Dauzère-Pérès and M. Sevaux. A Branch and Bound Method to Minimize the Number of Late Jobs on a Single Machine. *Research report 98/5/AUTO, Ecole des Mines de Nantes*, 1998.

[81] S. Dauzère-Pérès and M. Sevaux. An Efficient Formulation for Minimizing the Number of Late Jobs in Single-Machine Scheduling. *Research Report 98/9/AUTO, Ecole des Mines de Nantes*, 1998.

[82] M. Dell'Amico and S. Martello. *Linear Assignment*. Annotated Bibliographies in Combinatorial Optimization, John Wiley and Sons, 1997.

[83] E. Demeulemeester and W. Herroelen A Branch and Bound Procedure for the Multiple Resource-Constrained Project Scheduling Problem. *Management Science*, 38(12):1803–1818, 1992.

[84] E. Demeulemeester and W. Herroelen. New Benchmark Results for the Resource-Constrained Project Scheduling Problem. *Management Science*, 43:1485–1492, 1997.

[85] B. De Reyck and W. Herroelen. Assembly Line Balancing by Resource-Constrained Project Scheduling Techniques: A Critical Appraisal. *Technical Report, Katholieke Universiteit Leuven*, 1995.

[86] M. I. Dessouky, B. J. Lageweg, J. K. Lenstra and S. L. van de Velde. Scheduling identical jobs on uniform parallel machines. *Statistica Neerlandica*, 44:115–123, 1990.

[87] U. Dorndorf, E. Pesch and T. Phan-Huy. Solving the Open Shop Scheduling Problem. *Journal of Scheduling*, to appear.

[88] M. Drozdowski. *Selected Problems of Scheduling Tasks in Multiprocessor Computer Systems.* Instytut Informatyki Politechnika Poznań, 1997.

[89] M. Drozdowski and P. Dell'Olmo. Scheduling multiprocessor tasks for mean flow time criterion. *Computers and Operations Research* 27(6):571–585, 2000.

[90] J. Du and J. Y.-T. Leung. Complexity of scheduling parallel task systems. *SIAM Journal of Discrete Mathematics* 2:473–487, 1989.

[91] J. Du and J. Y-T. Leung. Minimizing Total Tardiness on One Machine is NP-Hard. *Mathematics of Operations Research* 15:483–495, 1990.

[92] J. Du J. Y.-T. Leung and C. S. Wong. Minimizing the number of late jobs with release time constraints. *Journal of Combinatorial Mathematics*, 11:97–107, 1992.

[93] L. Dupont. Ordonnancements sur machines à traitement par batch. *Research report GILCO Institu, fourn. National Polytechnique de Grenoble*, 1999.

[94] H. El Sakkout and M. Wallace. Probe Backtrack Search for Minimal Perturbation in Dynamic Scheduling. *Constraints*, 5(4):359–388, 2000.

[95] H. Emmons. One-machine sequencing to minimize certain functions of job tardiness. *Operations Research*, 17:701–715, 1969.

[96] J. Erschler. *Analyse sous contraintes et aide à la décision pour certains problèmes d'ordonnancement.* Thèse de Doctorat d'Etat, Université Paul Sabatier, 1976.

[97] J. Erschler, P. Lopez and C. Thuriot. Raisonnement temporel sous contraintes de ressource et problèmes d'ordonnancement. *Revue d'Intelligence Artificielle*, 5(3):7–32, 1991.

[98] P. Esquirol. *Règles et processus d'inférence pour l'aide à l'ordonnancement de tâches en présence de contraintes.* Thèse de l'Université Paul Sabatier, 1987.

[99] P. Esquirol, P. Lopez, H. Fargier and T. Schiex. Constraint Programming. *Belgian Journal of Operations Research*, 35(2):5–36, 1995.

[100] A. Federgruen and H. Groenevelt. Preemptive Scheduling of Uniform Machines by Ordinary Network Flow Techniques. *Management Science*, 32(3):341–349, 1986.

[101] A. J. Fernandez and P. M. Hill. A Comparative Study of Eight Constraint Programming Languages Over the Boolean and Finite Domains. *Constraints*, 5(3): 275-301, 2000.

[102] F. Focacci. *Solving Combinatorial Optimization Problems in Constraint Programming.* PhD Thesis, Università di Ferrara, 2001.

[103] F. Focacci, Ph. Laborie and W. Nuijten. Solving Scheduling Problems with Setup Times and Alternative Resources. *Proc. 5th International Conference on Artificial Intelligence Planning and Scheduling*, 2000.

[104] F. Focacci, A. Lodi, M. Milano and D. Vigo. Solving TSP through the integration of OR and CP techniques. *Proc. 4th International Conference on Principles and Practice of Constraint Programming*, 1998.

[105] F. Focacci and W. Nuijten. A Constraint Propagation Algorithm for Scheduling with Sequence Dependent Setup Times. *Proc. 2nd International Workshop on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, 2000.

[106] B. Fox. Chronological and Non-Chronological Scheduling. *Proc. 1st Annual Conference on Artificial Intelligence, Simulation and Planning in High Autonomy Systems*, 1990.

[107] G. Galambos and G. J. Woeginger. Minimizing the weighted number of late jobs in UET open shops. *Zeitschrift für Operations Research ZOR - Mathematical Methods of Operations Research*, 41:109–114, 1995.

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

[109] M. R. Garey, D. S. Johnson, B. B. Simons and R. E. Tarjan. Scheduling unit-time tasks with arbitrary release times and deadlines. *SIAM Journal of Computing*, 10:256–269, 1981.

[110] A. A. Gladky. On the complexity of minimizing weighted number of late jobs in unit time open shops. *Discrete Applied Mathematics*, 74:197–201, 1997.

[111] M. Gondran and M. Minoux. *Graphs and Algorithms.* John Wiley and Sons, 1984.

[112] T. Gonzalez. Unit Execution Time Shop Problems. *Mathematics of Operations Research*, 7:57–66, 1982.

[113] R. E. Graham, E. L. Lawler, J. K. Lenstra and A. H. G Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 4:287–326, 1979.

[114] C. Guéret. *Problèmes d'ordonnancement sans contrainte de précédence.* Thèse de l'Université de Technologie de Compiègne, 1997.

[115] C. Guéret, N. Jussien and C. Prins. Using intelligent backtracking to improve branch and bound methods: An application to open shop problems. *European Journal of Operational Research*, 127:344–354, 2000.

[116] W. D. Harvey and M. L. Ginsberg. Limited Discrepancy Search. *Proc. 14th International Joint Conference on Artificial Intelligence*, 1995.

[117] P. van Hentenryck. *Constraint Satisfaction in Logic Programming.* MIT Press, 1989.

[118] P. van Hentenryck. *The OPL Optimization Programming Language.* MIT Press, 1999.

[119] P. van Hentenryck, Y. Deville and C. M. Teng. A General Arc-Consistency Algorithm and its Specializations. *Artificial Intelligence*, 57(3):291–321, 1992.

[120] D.S. Hochbaum and D. Landy. Scheduling with batching: minimizing the weighted number of tardy jobs. *Operations Research Letters* 16:79–86, 1994.

[121] J. A. Hoogeveen, J. K. Lenstra and B. Veltman. Preemptive scheduling in a two-stage multiprocessor flow shop is NP-hard. *European Journal of Operational Research*, 89:172–175, 1997.

[122] W. Horn. Some simple scheduling problem. *Naval Research Logistics Quarterly*, 21:177–185, 1974.

[123] T. C. Hu. Parallel sequencing and assembly line problems. *Operations Research*, 9:841–848, 1961.

[124] J. Jaffar and J.-L. Lassez. Constraint Logic Programming. *Proc. 14th Annual ACM Symposium on Principles of Programming Languages*, 1987.

[125] J. Jaffar, S. Michaylov, P. J. Stuckey and R. H. C. Yap. The CLP(R) Language and System. *ACM Transactions on Programming Languages and Systems*, 14(3):339–395, 1992.

[126] K. Jansen and L. Porkolab. Preemptive parallel task scheduling in $O(n) + poly(m)$ time. *International Symposium on Algorithms and Computation, Taipeh*, 2000.

[127] S. M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quaterly*, 1:61–68, 1954.

[128] P. Kilby, P. Prosser and P. Shaw. A Comparison of Traditional and Constraint-based Heuristic Methods on Vehicle Routing Problems with Side Constraints. *Constraints*, 5(4):389–414, 2000.

[129] H. Kise, T. Ibaraki and H. Mine. A Solvable Case of the One-Machine Scheduling Problem with Ready and Due Times. *Operations Research*, 26(1):121–126, 1978.

[130] R. Kolisch, A. Sprecher and A. Drexel. Characterization and Generation of a General Class of Resource-Constrained Project Scheduling Problems. *Management Science*, 41(10):1693–1703, 1995.

[131] R. E. Korf. Improved Limited Discrepancy Search. *Proc. 13th National Conference on Artificial Intelligence*, 1996.

[132] R. Kowalski. Algorithm = Logic + Control. *Communications of the ACM*, 22(7):424–436, 1979.

[133] S. Kravchenko. On the complexity of minimizing the number of late jobs in unit time open shop. *Submitted to Discrete Applied Mathematics.*

[134] V. Kumar. Algorithms for Constraint Satisfaction Problems: A Survey. *AI Magazine*, 13(1):32–44, 1992.

[135] W. Kubiak, C. Sriskandarajah and K. Zaras. A Note on the Complexity of Openshop Scheduling Problems. *Information Systems and Operational Research*, 29:284–294, 1991.

[136] J. Labetoulle, E.L. Lawler, J.K. Lenstra and A.H.G. Rinnooy Kan. Preemptive scheduling of uniform machines subject to release dates. *Progress in combinatorial optimization*, Academic Press, pp. 245–261, 1984.

[137] Ph. Laborie. Modal Precedence Graphs and their Usage in ILOG Scheduler. *ILOG Optimization Internal Report OIR-1999-1*, 1999.

[138] Ph. Laborie. New Algorithms for Propagating Resource Constraints in AI Planning and Scheduling. *IJCAI-2001 Workshop on Planning with Resources* (submitted).

[139] F. Laburthe. Choco: implementing a CP kernel. *Proc. Workshop on Techniques for Implementing Constraint Systems (TRICS), 6th International Conference on Principles and Practice of Constraint Programming*, 2000.

[140] J.-C. Latombe. Failure Processing in a System for Designing Complex Assemblies. *Proc. 6th International Joint Conference on Artificial Intelligence*, 1979.

[141] J.-L. Laurière. A Language and a Program for Stating and Solving Combinatorial Problems. *Artificial Intelligence*, 10(1):29–127, 1978.

[142] E. L. Lawler. A Dynamic Programming Algorithm for Preemptive Scheduling of a Single Machine to Minimize the Number of Late Jobs. *Annals of Operations Research*, 26:125–133, 1990.

[143] E.L. Lawler. A pseudo-polynomial algorithm for sequencing jobs to minimize total tardiness. *Annals of Discrete Mathematics*, 1:331–342, 1977.

[144] E. L. Lawler, J. K. Lenstra, A. H. G. Rinnooy Kan and D. B. Shmoys. Sequencing and Scheduling: Algorithms and Complexity. *Operations Research and Management Science*, 4, 1989.

[145] E. L. Lawler and J. M. Moore. A functional equation and its application to resource allocation and sequencing problems. *Management Science*, 16:77–84, 1969.

[146] C.-Y. Lee and X. Cai. Scheduling one and two-processor tasks on two parallel processors. *IIE Transactions on Scheduling and Logistics*, 31:445–455, 1999.

[147] J. K. Lenstra, Alexander H.G. Rinnooy Kan and Peter Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[148] C. Le Pape. *Des systèmes d'ordonnancement flexibles et opportunistes.* Thèse de l'Université Paris XI, 1988.

[149] C. Le Pape. Implementation of Resource Constraints in ILOG Schedule: A Library for the Development of Constraint-Based Scheduling Systems. *Intelligent Systems Engineering*, 3(2):55–66, 1994.

[150] C. Le Pape and Ph. Baptiste. Resource constraints for preemptive job-shop scheduling. *Constraints*, 3:263–287, 1998.

[151] C. Le Pape and Ph. Baptiste. Heuristic Control of a Constraint-Based Algorithm for the Preemptive Job-Shop Scheduling Problem. *Journal of Heuristics*, 5:305–332, 1999.

[152] C. Le Pape, Ph. Couronné, D. Vergamini and V. Gosselin. Time-versus-Capacity Compromises in Project Scheduling. *Proc. 13th Workshop of the UK Planning Special Interest Group*, 1994.

[153] C. Le Pape and S. F. Smith. Management of Temporal Constraints for Factory Scheduling. In: C. Roland, F. Bodart and M. Léonard (editors), *Temporal Aspects in Information Systems*, North-Holland, 1988.

[154] M.-L. Lévy. *Méthodes par décomposition temporelle et problèmes d'ordonnan-cement.* Thèse de l'Institut National Polytechnique de Toulouse, 1996.

[155] O. Lhomme. Consistency Techniques for Numeric CSPs. *Proc. 13th International Joint Conference on Artificial Intelligence*, 1993.

[156] C.Y. Liu and R. L. Bulfin. Scheduling open shops with unit execution times to minimize functions of due dates. *Operations Research*, 36:553–559, 1998.

[157] H. C. R. Lock. An Implementation of the Cumulative Constraint. *Working Paper, University of Karlsruhe*, 1996.

[158] P. Lopez. *Approche énergétique pour l'ordonnancement de tâches sous contraintes de temps et de ressources.* Thèse de l'Université Paul Sabatier, 1991.

[159] P. Lopez, J. Erschler and P. Esquirol. Ordonnancement de tâches sous contraintes : une approche énergétique. *RAIRO Automatique, Productique, Informatique Industrielle*, 26(6):453–481, 1992.

[160] E. L. Lloyd. Concurrent task systems. *Operations Research* 29:189–201.

[161] A. K. Mackworth. Consistency in Networks of Relations. *Artificial Intelligence*, 8:99–118, 1977.

[162] K. Marriott and P. Stuckey. *Programming with Constraints: An Introduction.* MIT Press, 1998.

[163] P. D. Martin and D. B. Shmoys. A New Approach to Computing Optimal Schedules for the Job-Shop Scheduling Problem. *Proc. 5th Conference on Integer Programming and Combinatorial Optimization*, 1996.

[164] A. A. Mastor. An Experimental Investigation and Comparative Evaluation of Production Line Balancing Techniques. *Management Science*, 16(11):728–746, 1970.

[165] K. McAloon and C. Tretkoff. *Optimization and Computational Logic.* John Wiley and Sons, 1996.

[166] A. Mingozzi, V. Maniezzo, S. Ricciardelli and L. Bianco. An exact algorithm for project scheduling with resource constraints based on a new mathematical formulation. *Management Science*, 44:714–729, 1998.

[167] R. Mohr and T. C. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28:225–233, 1986.

[168] C. L. Monma and C. N. Potts. On the Complexity of Scheduling with Batch Setup Times. *Operations Research* 37:798–804, 1989.

[169] U. Montanari. Network of Constraints: Fundamental Properties and Applications to Picture Processing. *Information Sciences*, 7:95–132, 1974.

[170] J. M. Moore. An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs. *Management Science*, 15(1):102–109, 1968.

[171] E. Néron, Ph. Baptiste and J. N. D. Gupta. Solving Hybrid Flow Shop Problem Using Energetic Reasoning and Global Operations. *Research Report, University of Technology of Compiègne*, 2000.

[172] *Dictionary of Algorithms and Data Structures.* National Institute of standards and Technology, www.nist.gov/dads.

[173] W. P. M. Nuijten. *Time and Resource Constrained Scheduling: A Constraint Satisfaction Approach*. PhD Thesis, Eindhoven University of Technology, 1994.

[174] W. P. M. Nuijten. Private communication, 1995.

[175] W. P. M. Nuijten and E. H. L. Aarts. Constraint Satisfaction for Multiple Capacitated Job Shop Scheduling. *Proc. 11th European Conference on Artificial Intelligence*, 1994.

[176] W. P. M. Nuijten and E. H. L. Aarts. A Computational Study of Constraint Satisfaction for Multiple Capacitated Job-Shop Scheduling. *European Journal of Operational Research*, 90(2):269–284, 1996.

[177] W. P. M. Nuijten, E. H. L. Aarts, D. A. A. van Erp Taalman Kip and K. M. van Hee. Randomized Constraint Satisfaction for Job-Shop Scheduling. *Proc. AAAI-SIGMAN Workshop on Knowledge-Based Production Planning, Scheduling and Control*, 1993.

[178] W. P. M. Nuijten and C. Le Pape. Constraint-Based Job Shop Scheduling with ILOG SCHEDULER. *Journal of Heuristics*, 3:271–286, 1998.

[179] W. Nuijten and F. Sourd. New Time-Bound Adjustment Techniques for Shop Scheduling. *Proc. 7th International Workshop on Project Management and Scheduling*, 2000.

[180] L. Péridy. Le problème de job-shop : arbitrages et ajustements. *Thèse de l'Université de Technologie de Compiègne*, 1996.

[181] L. Péridy, Ph. Baptiste and E. Pinson. Branch and Bound Method for the Problem $1|r_i|\sum U_i$. *Proc. 6th International Workshop on Project Management and Scheduling*, 1998.

[182] M. Perregaard. *Branch and Bound Methods for the Multi-Processor Job-Shop and Flow-Shop Scheduling Problems*. MSc Thesis, University of Copenhagen, 1995.

[183] G. Pesant and M. Gendreau. A View of Local Search in Constraint Programming. *Proc. 2nd International Conference on Principles and Practice of Constraint Programming*, 1996.

[184] M. Pinedo. *Scheduling: Theory, Algorithms and Systems*. Prentice Hall, 1995.

[185] E. Pinson. *Le problème de job-shop*. Thèse de l'Université Paris VI, 1988.

[186] C.N. Potts, L.N. van Wassenhove. A decomposition algorithm for the single machine total tardiness problem. *Operations Research Letters*, 26:177–182, 1982.

[187] C. N. Potts and M. Y. Kovalyov. Scheduling with batching: A review. *European Journal of Operational Research* 120:228–249, 2000.

[188] P. Prosser. Hybrid Algorithms for the Constraint Satisfaction Problem. *Computational Intelligence*, 9(3):268–299, 1993.

[189] J.-F. Puget. A C++ Implementation of CLP. *Technical Report, ILOG*, 1994.

[190] J.-F. Puget and M. Leconte. Beyond the Glass Box: Constraints as Objects. *Proc. 12th International Symposium on Logic Programming*, 1995.

[191] J.-C. Régin. A filtering algorithm for constraints of difference in CSPs. *Proc. 12th National Conference on Artificial Intelligence*, 1994.

[192] J.-C. Régin. *Développement d'outils algorithmiques pour l'intelligence artificielle. Application à la chimie organique*. Thèse de l'Université Montpellier II, 1995.

[193] J.-C. Régin. Generalized Arc-Consistency for Global Cardinality Constraint. *Proc. 13th National Conference on Artificial Intelligence*, 1996.

[194] D. Rivreau. *Problèmes d'ordonnancement disjonctifs : règles d'élimination et bornes inférieures*. Thèse de l'Université de Technologie de Compiègne, 1999.

[195] R. Rodosek and M. Wallace. A Generic Model and Hybrid Algorithm for Hoist Scheduling Problems. *Proc. 4th International Conference on Principles and Practice of Constraint Programming*, 1998.

[196] R. Rodosek, M. Wallace and M. Hajian. A New Approach to Integrating Mixed Integer Programming and Constraint Logic Programming. *Annals of Operations Research*, 86:63–87, 1999.

[197] S. Sahni. Preemptive Scheduling with Due Dates. *Operations Research* 27:925–934, 1979.

[198] S. Sahni and Y. Cho. Scheduling Indepensant Tasks with Due Times on a Uniform Processor System. *Journal of the Association for Computing Machinery*, 27:550–563, 1980

[199] D. F. Shallcross. A polynomial algorithm for a one machine batching problem. *OR Letters* 11:213–218, 1992.

[200] P. Shaw. Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems. *Proc. 4th International Conference on Principles and Practice of Constraint Programming*, 1998.

[201] B. Simons. Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines. *SIAM Journal of Computing*, 12:294–299, 1983.

[202] S. F. Smith. OPIS: A Methodology and Architecture for Reactive Scheduling. In: M. Zweben and M. Fox (editors), *Intelligent Scheduling*, Morgan Kaufmann, 1994.

[203] S. F. Smith and C.-C. Cheng. Slack-Based Heuristics for Constraint Satisfaction Scheduling. *Proc. 11th National Conference on Artificial Intelligence*, 1993.

[204] F. Sourd. *Contribution à l'étude et à la résolution de problèmes d'ordonnan-cement disjonctif.* Thèse de l'Université Paris VI, 2000.

[205] F. Sourd and W. Nuijten. Multiple-Machine Lower Bounds for Shop Scheduling Problems. *INFORMS Journal on Computing*, 12(4):341–352, 2000.

[206] R. M. Stallman and G. J. Sussman. Forward Reasoning and Dependency-Directed Backtracking in a System for Computer-Aided Circuit Analysis. *Artificial Intelligence*, 9(2):135–196, 1977.

[207] G. L. Steele Jr. *The Definition and Implementation of a Computer Programming Language Based on Constraints*. PhD Thesis, Massachusetts Institute of Technology, 1980.

[208] W. Szwarc, F. Della Croce and A. Grosso. Solution of the single machine total tardiness problem. *Journal of Scheduling*, 2:55–71, 1999.

[209] V. G. Timkovsky. Is a unit-time job shop not easier than identical parallel machines ? *Discrete Applied Mathematics* 85:149-162, 1998.

[210] V. G. Timkovsky. Identical parallel machines vs. unit-time Shops, preemptions *vs.* chains, and other offsets in scheduling complexity. *Technical report, Star Data Systems Inc.*, 1998.

[211] Ph. Torres and P. Lopez. On Not-First/Not-Last conditions in disjunctive scheduling. *European Journal of Operational Research*, 127:332–343, 2000.

[212] E.P.K. Tsang. *Foundations of Constraint Satisfaction.* Academic Press, 1993.

[213] R. J. M. Vaessens, E. H. L. Aaarts and J. K. Lenstra. Job-Shop Scheduling by Local Search. *INFORMS Journal on Computing*, 8:302–317, 1996.

[214] C. Varnier, P. Baptiste and B. Legeard. Le traitement des contraintes disjonctives dans un problème d'ordonnancement : exemple du Hoist Scheduling Problem. *Actes 2èmes journées francophones de programmation logique*, 1993.

[215] M. Wallace, S. Novello and J. Schimpf. ECLiPSe: A Platform for Constraint Logic Programming. *IC-Parc, Imperial College, London*, 1997.

[216] T. Walsh. Depth-bounded Discrepancy Search. *Proc. 15th International Joint Conference on Artificial Intelligence*, 1997.

[217] S. Webster and K.R. Baker. Scheduling Groups of Jobs on a Single Machine. *Operations Research* 43:692–703, 1995.

Index