Contributions of Inclusion-Exclusion to exact or approximate solution of scheduling problems

Olivier Ploton

UNIVERSITÉ DE TOURS

ÉCOLE DOCTORALE MIPTIS

Laboratoire d'Informatique (LIFAT, EA 6300)

THÈSE présentée par :

Olivier PLOTON

soutenue le : **25 novembre 2022**

pour obtenir le grade de : **Docteur de l'Université de Tours**

Discipline/Spécialité : INFORMATIQUE

Apports de l'Inclusion-Exclusion à la résolution exacte ou approchée de problèmes d'ordonnancement

THÈSE DIRIGÉE PAR :

M. T'KINDT Vincent	Professeur des Universités, Université de Tours

RAPPORTEURS :

M. LIEDLOFF Mathieu	Maître de conférences HDR, Université d'Orléans
Mme MUNIER-KORDON Alix	Professeure des Universités, Sorbonne Université

PRÉSIDENTE :

Mme BRAUNER-VETTIER Nadia	Professeure des Universités, Université Grenoble Alpes

JURY :

Mme BAZGAN Cristina	Professeure des Universités, Université Paris-Dauphine
M. DELLA CROCE Federico	Professeur, Politecnico di Torino, Italie
M. GIROUDEAU Rodolphe	Maître de conférences HDR, Université de Montpellier

À Stéphanie et Marie.

Remerciements

Les études forment un long cheminement parsemé d'incertitudes et de rebondissements. Chaque étape, chaque projet est une aventure en soi, partagée entre un étudiant et son professeur. Il arrive que la transmission du savoir du professeur à l'étudiant s'opère de manière tellement naturelle que la motivation s'en trouve décuplée et permette des efforts et des réussites inimaginables.

En tant qu'étudiant j'ai vécu cette situation idéale par deux fois. La première, c'était avec mon professeur de mathématiques d'hypotaupe, Michel Quercia, le professeur le plus époustouflant que j'aie connu, parmi une concurrence pourtant très relevée. La seconde, c'est avec Vincent T'kindt au cours de cette thèse. Je mesure toute l'audace dont il a fait preuve en me prenant en thèse, et tout le respect qu'il m'a témoigné au cours de ces trois années ponctuées d'efforts et de difficultés, d'espoirs et de déceptions, d'avancées et de circonvolutions. J'en suis très honoré.

J'ai effectué cette thèse tout en enseignant les Mathématiques et l'Informatique en tant que PRAG à L'université de Tours. Cette thèse est donc auto-financée, mais j'ai bénéficié de décharges de service pour un total équivalent à une année complète. Je remercie le Conseil Académique de m'avoir accordé ces avantages matériels, et par là même de m'avoir accordé sa confiance.

Je remercie Hubert Cardot, Emmanuel Humbert et Sandrine Dallet-Choisy de m'avoir appuyé auprès du Conseil Académique et de m'avoir constamment soutenu.

Je remercie mes collègues Boris Andreianov, Thierry Brouard et Florent Malrieu d'avoir créé un contexte favorable à cette thèse par leurs conseils avisés.

Je remercie Hélène Tison pour sa patience, ainsi que tous les membres de l'équipe de recherche ROOT pour leur bienveillance à mon égard : Jean-Charles Billaut, Ronan Bocquillon, Jean-Paul Chemla, Pierre Desport, Yannick Kergosien, Christophe Lenté, Patrick Martineau, Emmanuel Néron, Tifenn Rault, Romain Raveaux, Ameur Soukhal.

Je remercie Mathieu Liedloff et Alix Munier-Kordon d'avoir accepté d'être rapporteurs de ma thèse, de consacrer du temps et de l'énergie à relire mon manuscrit et à formuler des remarques constructives. Je remercie également Cristina Bazgan, Nadia Brauner-Vettier, Federico Della Croce et Rodolphe Giroudeau de l'intérêt qu'ils témoignent à ce sujet de thèse en acceptant d'être membres du jury.

Résumé

Dans cette thèse, nous nous intéressons à la résolution de problèmes d'ordonnancement par Inclusion-Exclusion. La formule d'Inclusion-Exclusion est une formule de combinatoire, encore peu utilisée en ordonnancement. Appliquée à des données informatiques, elle permet de dénombrer, par une somme comportant un nombre exponentiel de termes, le nombre de solutions de problèmes de couverture ou de permutation, et par contrecoup d'expliciter des solutions optimales de tels problèmes.

D'un point de vue théorique, notre principale contribution est de démontrer qu'une grande classe de problèmes d'ordonnancement peut se résoudre, par Inclusion-Exclusion, à l'optimalité et avec une complexité au pire cas modérément exponentielle en temps et pseudopolynomiale en espace. Ce résultat s'applique à tout problème d'ordonnancement à machines parallèles ainsi qu'à tout problème d'atelier à cheminement unique, en présence de n'importe quelle contrainte temporelle d'intervalle, et prend en compte n'importe quel objectif régulier du type coût maximum ou coût total.

La formule d'Inclusion-Exclusion permet de simplifier des problèmes en relâchant leur contrainte de couverture, mais nécessite de résoudre un nombre exponentiel de problèmes relâchés. D'un point de vue pratique, nous étudions une piste pour bénéficier des avantages de l'Inclusion-Exclusion sans en supporter les inconvénients. Nous décrivons une méthode itérative, exploitable dans un algorithme de branchement, pour minorer l'objectif optimum d'un problème de permutation, fondée uniquement sur la résolution de problèmes relâchés.

Mots clefs : ordonnancement, complexité au pire cas, méthodes exactes, approximation, Inclusion-Exclusion.

Abstract

In this thesis, we are interested in solving scheduling problems by Inclusion-Exclusion. The Inclusion-Exclusion formula is a combinatorial formula, still not much used in scheduling. Applied to computer science, it enables to count, by means of a sum involving an exponential number of terms, the number of solutions of coverage or permutation problems, and in turn to build explicit optimal solutions of such problems.

From a theoretical point of view, our main contribution is to demonstrate that a large class of scheduling problems can be solved to optimality by Inclusion-Exclusion, and with a moderate-exponential worst-case time complexity and a pseudopolynomial worst-case space complexity. This result applies to any parallel-machine scheduling problem as well as any permutation flowshop problem, in the presence of any interval time constraint, and takes into account any regular maximum or total cost objective.

The Inclusion-Exclusion formula enables to simplify problems by relaxing their coverage constraint, but it requires solving an exponential number of relaxed problems. From a practical point of view, we are studying a way to benefit from the advantages of Inclusion-Exclusion without bearing the disadvantages. We describe an iterative method, usable in a branching algorithm, to compute a lower bound on the optimum objective of a permutation problem, based only on the solution of relaxed problems.

Keywords: scheduling, worst case complexity, exact methods, approximation, Inclusion-Exclusion.

Contents

Illustrations

Introduction

This thesis deals with Inclusion-Exclusion for scheduling. More precisely, it establishes a link between these three fields, which we describe hereafter: Scheduling, Exponential Algorithms and Combinatorics.

Scheduling is the art of matching over time jobs to be processed with resources to process them, in order to optimize an objective. Scheduling problems have major practical applications in, e.g., production systems, information systems, and project management. Scheduling is a subfield of Operations Research, which consists in analyzing and solving problems in order to help decision-making and management. Scheduling problems are very challenging, and often intractable, in a way we are about to precise.

Exponential Algorithmics aims to precise the notion of intractability. On a theoretical point of view, the hardness of an algorithm is characterized by its worst-case complexity, *i.e.* the maximum consumption of resources (e.g. time or space) with respect to the size of its data, and the intrinsic hardness of a problem is the hardness of the best algorithm to solve it. Under well-known and widely accepted conjectures (namely the $P \neq NP$ conjecture, see Cook [17] and Levin [51], and the Exponential Time Hypothesis by Impagliazzo and Paturi [38]), many scheduling problems are intrinsically exponential in time. Exponential Algorithmics aims to design an algorithm to solve such a problem with the lowest possible, though exponential, worst-case time complexity.

Combinatorics is the art of counting mathematical objects. This field of mathematics has tight connections with probabilities, number theory, algebra, group theory, and graph theory, one of the mathematical foundations of Operations Research. Combinatorics brings together a set of very varied techniques: binomial coefficients and related notions, generating series, convolution and Fourier analysis, convexity and entropy theory, Möbius inversion, and Inclusion-Exclusion.

Inclusion-Exclusion is a combinatorial formula, dating back to the 18th century. In its mathematical form, it expresses the cardinal of a union of sets given the cardinals of their partial intersections. Its application to computer science enables to count how many instances of a given data structure may be built without explicitly building them, and constitutes a means to avoid brutal enumeration.

Inclusion-Exclusion has a very special and unusual connection with computer science: usually, algorithms do not use combinatorics to compute their results, and combinatorics is used only to analyze algorithms and count the number of elementary operations they require, thus deriving their complexities. On the contrary, the Inclusion-Exclusion formula is concretely computed inside algorithms, in order to derive the number of solutions of a problem and therefore to decide whether this problem is solvable or not.

Historically, Inclusion-Exclusion was first applied to computer science independently by Kohn et al. [43], Karp [42], and Bax [5], essentially to solve the Traveling Salesman Problem. Later on, Inclusion-Exclusion enabled Bjorklund and Husfeldt [8], and independendly Koivisto [44] to achieve a major improvement in the solution of the Maximum Independent Set problem. Thus, Inclusion-Exclusion gained in popularity, and several books, e.g. Fomin and Kratsch [23], dedicated a whole chapter to this technique. Nederlof [59, 60, 61, 62] described many algorithms and showed the theoretical interest of this technique for Operations Research.

Despite its new popularity for Operations Research, Inclusion-Exclusion is not yet much used for scheduling. The purpose of this thesis is to study how the field of scheduling can benefit from the Inclusion-Exclusion technique.

Contributions

As a first contribution, we identify a very common way of applying Inclusion-Exclusion to a scheduling problem: we apply Inclusion-Exclusion to sets of jobs, relax the problem accordingly, and solve each relaxed problem by dynamic programming. This requires computing many correlated dynamic programming schemes. We introduce a new technique, zero sweeping, to exploit correlations between dynamic programming schemes and compute all schemes together faster than separately.

As a second contribution, we prove that a wide class of scheduling problems, including any parallel-machine or flowshop problem with any deadline and release date constraint and any regular objective, can be solved to optimality with a moderate-exponential worst-case time complexity combined with a pseudopolynomial worst-case space complexity, using Inclusion-Exclusion. This contribution enhances the state of the art in two ways: first, our result is very general, whereas many state-of-the-art algorithms are specialized to a very specific subproblem. Second, most state-of-the-art algorithms achieve a moderate exponential-time complexity at the expense of an exponential-space complexity, whereas the algorithms we propose keep a pseudopolynomial space complexity.

As a third contribution, we establish a link between Inclusion-Exclusion and Lagrangian relaxation based on job penalties. For a permutation problem, where the aim is to determine in which order jobs must be scheduled, we describe a new iterative method to compute a better approximation of the dual Lagrangian optimum objective than the traditional subgradient descent method, therefore deriving a better lower bound of the optimum objective.

Outline

In chapter 1, we introduce the notations of scheduling, in particular Gantt [26] charts and Graham's [32] three-field notation, and we describe several scheduling environments with examples. Then, we present computational complexity and related notions, and we review the complexity classes of several scheduling problems. Finally, we review the state-of-the-art techniques of exponential-time algorithmics and their application to scheduling problems. We emphasize the interest of exponential algorithmics to solve scheduling problems to optimality.

In chapter 2, we describe the principles of Inclusion-Exclusion, and we explain how and in which circumstances Inclusion-Exclusion enables to save space and enhance worst-case space complexity without degrading worst-space time complexity. We review several techniques to accelerate Inclusion-Exclusion, from a theoretical and practical point of view: Bonferroni inequalities, Möbius inversion, Abstract Tubes, Index space trimming, and finally we contribute to a new one: zero sweeping.

In chapter 3, we provide a generic algorithm for solving unrelated parallel machine scheduling problems in presence of job release dates and deadlines, to minimize general regular maximum and sum objective functions. Thanks to the use of Inclusion-Exclusion combined with generating series and convolution, this algorithm runs with moderate exponential time and pseudopolynomial space worst-case complexities, and enhances the state of the art.

In chapter 4, we provide another Inclusion-Exclusion based generic algorithm for solving permutation flowshop scheduling problems in presence of job release dates and deadlines, to minimize general regular maximum and sum objective functions. Again, this algorithm runs with moderate exponential time and pseudopolynomial space worst-case complexities, and enhances the state of the art. Several extensions of this algorithm are discussed.

In chapter 5, we study a way to benefit from the advantages of Inclusion-Exclusion, which enables to relax and thus simplify a problem, without bearing the inconveniences. We express, in essence, Inclusion-Exclusion through Lagrangian penalties, instead of expressing it with a sum across subsets. We contribute to a new iterative method to derive a lower bound of the optimal objective of a permutation scheduling problem based only on the solution of the relaxed problem.

Chapter 1

Scheduling and Exponential Algorithms

Topics

- We describe the basics and the notations of scheduling.
- We recall the notions of computational complexity, complexity classes and polynomial reduction between problems.
- We review the complexity classes of many well-known scheduling problems, along with their reduction graph.
- We review the state-of-the-art techniques of exponential-time algorithmics and their application to scheduling problems.
- These techniques show the interest of exponential algorithmics to solve scheduling problems to optimality.

1.1 Scheduling Problems

The domain of scheduling aims at modeling and optimizing problems such as the management of projects, of stocks, the organization of a production line, the management of transport or energy networks.

Scheduling is involved when some work has to be done using some resources and when there are several possible organizations to achieve it. The elementary units of work are called operations, and the resources are generically called machines, even if they correspond to human workers. Scheduling aims at finding the best allocation of machines to process operations in order to minimize a cost, called the objective.

We introduce basic scheduling notions, as defined in e.g. Pinedo [64] or Brucker [12].

1.1.1 Basics of Scheduling

An operation is an elementary amount of work to be done. It has to be processed at once by a machine. A job is the user-level notion of work to be done. Depending on the context, it is either a single operation, or a group of operations.

There are n jobs to be scheduled on m machines. Each job is labeled by an integer $j \in \{1, \ldots, n\}$, and each machine is labeled by an integer $i \in \{1, \ldots, m\}$.

Operations of job j are labeled using j preceded by any useful index, e.g. $O_{1j}, O_{2j}, \ldots$ or O_j in the case of a single-operation job. For the sake of simplicity, we describe notations in the case where jobs are single-operation, so operations are written O_j. The notations we define are straightforward to extend to the case of jobs with multiple operations.

It takes some time for a resource to process an operation. This duration is called the processing time p_j of the operation O_j. All times and durations are non-negative integers and are expressed in an arbitrary time unit. The number of jobs and machines, and the processing times are always implicitly part of the instance of a scheduling problem.

As a result, a schedule S is defined by the placement of all operations. The placement of operation O_j is defined by the machine $I_j(S)$ which actually processes it and by its completion time $C_j(S)$. When there is no ambiguity, it is usual to remove the reference to S in the notation and thus to write I_j and C_j. The start time of an operation O_j is most often derived from its completion time: it is $(C_j - p_j)$.

Graphically, a schedule is represented via a Gantt chart [26], a chronogram in which each line corresponds to a machine and each column corresponds to a time unit. Figure 1.1 shows a sample Gantt chart with 4 jobs on 2 machines.

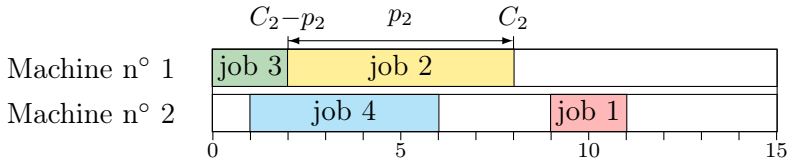


Figure 1.1: A sample Gantt chart

On a timeline, operation O_j corresponds to the interval $[C_j - p_j, C_j[$ or actually to its integer counterpart $\mathbb{N} \cap [C_j - p_j, C_j[$. As we always restrict to integer times and durations, we hereafter write $[a, b[$ as an abbreviation for the integer interval $[a, b[= \{a, \ldots, b-1\}$. Notice the use of a semi-open interval, which makes notations easy and consistent with their real counterpart. With these notations, all interval operations are easy to express:

- Countings elements: for $a \leqslant b$, $[a, b[= \{a, \ldots, b-1\}$ contains exactly $b - a$ elements.
- Splitting an interval: for $a \leqslant b \leqslant c$, we can split $[a, c[$ into the disjoint union $[a, c[= [a, b[\uplus [b, c[$
- Checking whether two intervals are disjoint: for $a \leqslant b$ and $c \leqslant d$, we have $[a, b[\cap [c, d[= \varnothing \iff b \leqslant c \ \vee \ a \geqslant d$.

We now define the notion of precedence relation between operations. An operation O_j is said to precede another operation $O_{j'}$, which we write $O_j \prec O_{j'}$, when O_j is already completed when $O_{j'}$ starts. That is:

$$O_j \prec O_{j'} \iff C_j \leqslant C_{j'} - p_{j'} \tag{1.1}$$

Precedence constraints between operations can be deduced from a particular situation, *i.e.* specified within an instance, or they can be intrinsic to a problem, *i.e.* specified by the structure of the problem. In the latter case, they are part of the so-called routing constraints.

More generally, a problem stipulates constraints of any nature on operations. A schedule is said to be admissible when all problem constraints are answered by its operations. We always consider scheduling models where a machine can process only one operation at a time. So, every admissible schedule must answer the so-called disjunctive constraints: intervals representing operations in a Gantt chart can not overlap, hence must be disjoint. Formally, for two distinct operations $O_j \neq O_{j'}$ scheduled on the same machine (*i.e.* $I_j = I_{j'}$), we must have:

$$[C_j - p_j, C_j[\cap [C_{j'} - p_{j'}, C_{j'}[= \varnothing$$

The Graham Notation

The field of scheduling is extremely vast, and, from this point of view, is comparable to the field of partial differential equations. A partial differential equation is very simple to express a general way: given a function f, consider its derivatives with respect to some variables, usually space (x, y, z) and time (t), and require that a particular expression involving certain derivatives of f is null: $X(f, \frac{\partial f}{\partial x}, \ldots, \frac{\partial f}{\partial t}, \frac{\partial^2 f}{\partial x^2}, \frac{\partial^2 f}{\partial x \partial y}, \ldots) = 0$. Hence, the description of a differential problem reduces to a single equation. But this conciseness hides very different solution techniques on a case by case basis. A single particular equation can constitute an area of research, e.g. Maxwell and electromagnetism, or Navier-Stokes and fluid mechanics.

The same is true for scheduling. Consequently, schedulers extensively use the notation of Graham [32] to refer to particular scheduling problems. It is a three-field notation $\alpha|\beta|\gamma$, where α describes the machine environment, β describes extra constraints, and γ presents the objective functions or costs to minimize.

Table 1.2 p. 22 summarizes usual values for these three fields, but we describe them precisely in sections 1.1.2 p. 22, 1.1.3 p. 24, 1.1.4 p. 25. There are many other possible values described in the literature.

Table 1.2: Some usual values of Graham's notation $\alpha|\beta|\gamma$

α machine environment	β extra constraints	γ objective to minimize
1: single machine	r_j: release time	$\gamma = \max_j f_j$ or $\gamma = \sum_j w_j f_j$
P: identical parallel machines	$\tilde{d}_j$: deadline	w_j: weight
R: unrelated parallel machines	d_j: due date	f_j: individual cost
	prec, *chains*: precedences	C_j: completion time
F: (non-permutation) flowshop	*prmu*: permutation (flowshop)	L_j: lateness
J: jobshop	*pmtn*: preemption	T_j: tardiness
O: openshop	*nowait*: no waiting time	U_j: tardiness indicator
	noidle: no idle time	E_j: earliness

1.1.2 Machine environments

In Graham's notation of a scheduling problem, the field α describes the machine environment. There are basically two families of machine environments: parallel machine environments ($\alpha \in \{1, P, R\}$), including the single machine environment $\alpha = 1$ as a particular case, and shop environments ($\alpha \in \{F, J, O\}$). Whenever the number of machines m is fixed, its value is appended to the environment α, as in e.g. $P2$ or $F3$.

Parallel machine environments

In parallel machine environments, each job has a single operation, which can be scheduled on any machine. Such environments notably include:

Single-machine (1): there is a single machine, *i.e.* $m = 1$. This environment is denoted by 1 rather than by $P1$. This is in fact a sequential environment.

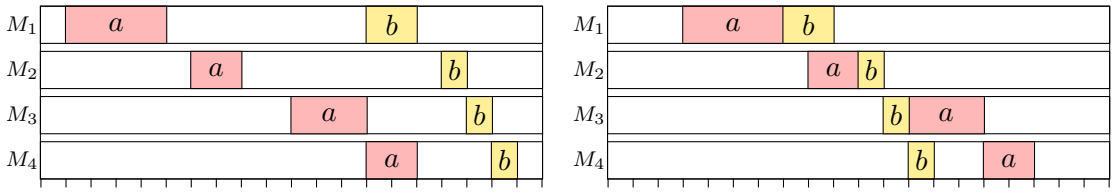
Identical parallel machines (P): there are m machines with identical features, so the processing time p_j of a job j and all other parameters do not depend on the machine i which processes the job.

Unrelated parallel machines (R): each machine has its own features, so the processing time p_{ij} of a job depends not only on the job j but also on the machine i. This may also be the case for other parameters.

Shop environments

In these environments, there are m machines i and n jobs j, each one being divided into m operations O_{ij}: operation O_{ij} is the part of job j which must be processed by machine i. So, each job is processed by the m machines, and each machine processes one operation of each job. We distinguish between shop environments based on the order in which operations have to be processed, *i.e.* based on the routing constraints.

In flowshop environments, each job is processed successively by machines $1, 2, \ldots, m$, in this order. That is, there is a precedence relation, called the routing, between operations of each job j: $O_{1j} \prec O_{2j} \prec \ldots \prec O_{mj}$. Graphically, operations of each job form a sort of diagonal on a Gantt chart, as shown on Figure 1.3 p. 23.



(a) Permutation flowshop $F|prmu, \beta|\gamma$: jobs cannot overtake eachother

(b) Non-permutation flowshop $F|\beta|\gamma$: jobs can overtake eachother

Figure 1.3: Flowshop environments

The simplest flowshop environment is the permutation flowshop. In this environment, all machines i process the operations O_{ij} in the same job order: $O_{i,j_1}, O_{i,j_2}, \ldots, O_{i,j_n}$. Each schedule is therefore characterized by a common order, or common permutation: $\pi = (j_1, j_2, \ldots, j_n)$. This permutation is not imposed in the instance and does not correspond to a set of precedence constraints. On the opposite, the aim is usually to find a permutation π which minimizes the objective.

Since the order of jobs is the same on each machine, jobs can not overtake each other from one machine to the other. Graphically, diagonals formed by operations of each job on a Gantt chart can not be interlaced, as shown on Figure 1.3a. The permutation flowshop environment is denoted by $F|prmu$. The indication "$prmu$" is written as a constraint, but permutation flowshop $F|prmu$ is specific and conceptually distinct from non-permutation flowshop.

The (non-permutation) flowshop environment is denoted by F. In this environment, each machine processes operations in its own job order: $O_{i,j_{i1}}, O_{i,j_{i2}}, \ldots, O_{i,j_{in}}$. That is, each machine i has its own permutation $\pi_i = (j_{i1}, j_{i2}, \ldots, j_{in})$, and the aim is usually to find a tuple of permutations $(\pi_1, \ldots, \pi_m)$ which minimizes the objective. Jobs can overtake each other from one machine to the other. Graphically, diagonals formed by operations of each job on a Gantt chart can be interlaced, as shown on Figure 1.3b.

In jobshop and openshop environments, jobs are not necessarily processed by machines 1 to m in this order. In the jobshop environment, denoted by J, each job j has its own routing. That is, the instance contains n imposed precedence chains of the form $O_{i_{j1},j} \prec O_{i_{j2},j} \prec \ldots \prec O_{i_{jm},j}$. The non-permutation flowshop environment F is a particular case of the jobshop environment J, where precedence chains have the same form for all jobs, *i.e.* when $\forall\, k \in \{1, \ldots, m\}, i_{jk} = i_k$, and we can take $i_k = k$ without loss of generality.

In the openshop environment O, there are no routing constraints between operations of the same job. Figure 1.4 p. 24 shows an admissible schedule with 2 jobs $j \in \{a, b\}$ for an openshop problem. It is also an admissible schedule for a jobshop problem with routings $O_{1a} \prec O_{4a} \prec O_{2a} \prec O_{3a}$ and $O_{2b} \prec O_{3b} \prec O_{4b} \prec O_{1b}$.

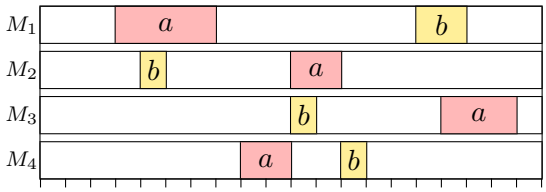


Figure 1.4: Job shop and Open shop environments

1.1.3 Additional constraints

In Graham's notation of a scheduling problem, the field β describes additional constraints. Some of them reduce the set of admissible schedules. Some others may signal a particular case, modify a problem or even relax other constraints. We describe them hereafter:

Release date (r_j): Operation O_j cannot start before the date r_j.

Deadline ($\tilde{d}_j$): Operation O_j cannot complete after the date $\tilde{d}_j$.

Together, release date and deadline form interval constraints: the processing interval of operation O_j must be included in the interval defined by the constraint: $[C_j - p_j, C_j[\subset [r_j, \tilde{d}_j[$.

Due date (d_j): Operation O_j may complete after date d_j, but in this case the job is late which may induce a penalty. So, this indication is not a constraint, it is a supplementary data of the instance used to compute some objectives (lateness, tardiness, tardiness indicator, see section 1.1.4).

Precedence (*prec*): Some operations must precede others. Generally, precedences are expressed as a directed acyclic graph (DAG). The *chains* constraint corresponds to the case where the precedence graph is a disjoint union of chains of the form $O_{j_1} \prec O_{j_2} \prec \ldots \prec O_{j_\ell}$.

Preemption (*pmtn*): Normally, operations are atomic or indivisible. In this setting, an operation may be interrupted and resumed later, without penalty. So, operations may be divided into fragments. This setting is often used to compute lower bounds for non-preemptive problems.

No idle time (*noidle*): No machine may be idle until all its operations have been scheduled. That is, there can be no delay between two consecutive operations processed by a machine i.

No wait time (*nowait*): There cannot be any delay between two consecutive operations of a same job. So, if two operations $O_{1,j}$ and $O_{2,j}$ of a same job j are consecutive, we must have $C_{1,j} = C_{2,j} - p_{2,j}$.

Custom constraints Any particular constraint can be specified as a condition. For example, the constraint "$p_j = p$" actually means $\forall\, j, p_j = p$, *i.e.* all processing times are equal, and their common value is p.

1.1.4 Objective Functions

In Graham's notation, the field γ describes the objective function. The objective function γ usually computes a positive integer cost based on a schedule. The aim is to minimize this cost among all admissible schedules.

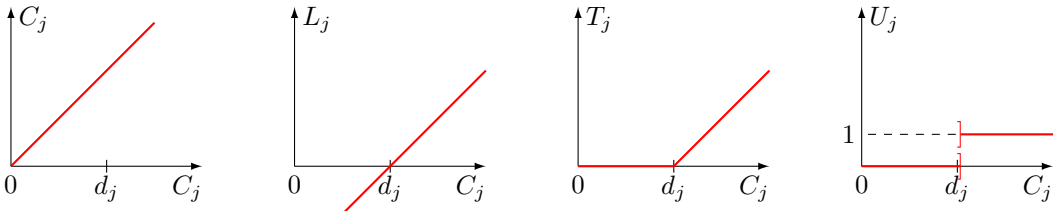
It is possible to specify a trivial objective function, whose result is a constant. It is denoted by the symbol "-". In this case, there is nothing to optimize, the problem just consists in finding an admissible schedule. In other cases, although there is no strict convention, an objective function γ is most often computed as the maximum or weighted sum of elementary costs $f_j \geqslant 0$, with weights $w_j \geqslant 0$. We describe hereafter how:

The elementary cost of an operation O_j is systematically computed from its completion time. So, given a schedule S, the elementary cost of O_j is $f_j(C_j(S))$. The objective value $\gamma(S)$ is then computed using one of these three formulas:

$$\begin{array}{rlll} \text{maximum:} & \gamma(S) = \max_{j \in S} f_j(C_j(S)) & \text{abbreviated into} & \gamma = f_{\max} \\ \text{sum:} & \gamma(S) = \sum_{j \in S} f_j(C_j(S)) & \text{abbreviated into} & \gamma = \sum f_j \\ \text{weighted sum:} & \gamma(S) = \sum_{j \in S} w_j f_j(C_j(S)) & \text{abbreviated into} & \gamma = \sum w_j f_j \end{array}$$

As an extension, it is possible to assign an infinite cost to an operation, *i.e.* to declare that $f_j(C_j) = +\infty$ for some C_j. Such an assignation prevents operation O_j from ending exactly at date C_j, because if $f_j(C_j(S)) = +\infty$ then $\gamma(S) = +\infty$ and S cannot be an optimal schedule.

Figure 1.5 represents the most usual elementary costs. These costs are non-decreasing with respect to completion times C_j, and are said to be regular. Several elementary costs f_j are computed from a due date, denoted by d_j and described in section 1.1.3.



(a) Completion time: C_j

(b) Lateness: $L_j = C_j - d_j$

(c) Tardiness: $T_j = \max(L_j, 0)$

(d) Unit cost: $U_j = 1 \Leftrightarrow T_j > 0$

Figure 1.5: Some usual regular elementary cost functions

Regular elementary costs include:

Completion time (C_j): It is at the same time an integer (used as parameter of all elementary cost functions) and a cost function (actually the identity function).

Lateness (L_j): It is defined as $L_j = C_j - d_j$. Its value can be negative (operation O_j is early or exactly on time) or non-negative (operation O_j is tardy).

Notice that an objective function is usually required to be non-negative. The lateness does not meet this requirement, but it is easy to transform a lateness-based objective into a non-negative objective: it suffices to add a large enough constant K, or equivalently to replace each d_j with $d_j - K$.

Tardiness (T_j): It is defined as $T_j = \max(L_j, 0) = C_j - d_j$, if $C_j > d_j$; 0 otherwise. Its value can be null (operation O_j is early or exactly on time) or non-negative (operation O_j is tardy).

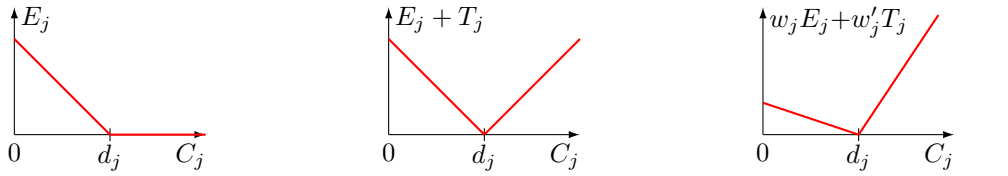
Tardiness Indicator or Unit cost (U_j): It is defined as $U_i = 1$, if $C_j > d_j$; 0 otherwise.

Some combinations, as $T_{\max}$ and $U_{\max}$ are irrelevant: we have $T_{\max} = \max(0, L_{\max})$ and $U_{\max} = 1$ if $L_{\max} > 0$, 0 otherwise. Their optimal value and the corresponding optimal schedules are derived from the optimization of $L_{\max}$. Actually, in the very common case where $L_{\max}$ is guaranteed to be positive, $T_{\max}$ is equal to $L_{\max}$, and referring to either objective name is a matter of taste. Many scheduling articles (e.g. Jackson [39] about the famous Jackson's rule) refer to $T_{\max}$, but in this document we choose to refer to $L_{\max}$. The combination $\sum_j L_j$ is not relevant either: minimizing $\sum_j L_j$ is equivalent to minimizing $\sum_j C_j$ (up to the additive constant $\sum_j d_j$). The same rationale applies for the weighted sum. Finally, there are 8 basic combinations summarized in table 1.6:

Table 1.6: Most basic regular objective functions

objective	name
$C_{\max}$	makespan
$L_{\max}$	maximum lateness
$\sum C_j$	total completion time
$\sum w_j C_j$	total weighted completion time
$\sum T_j$	total tardiness
$\sum w_j T_j$	total weighted tardiness
$\sum U_j$	number of tardy jobs
$\sum w_j U_j$	weighted number of tardy jobs

We now define the earliness E_j as: $E_j = \max(-L_j, 0) = C_j - d_j$ if $C_j > d_j$, 0 otherwise. Contrary to the elementary costs $f_j \in \{C_j, L_j, T_j, U_j\}$, E_j is not a non-decreasing function of C_j and it is said to be non-regular. Earliness is used for just in time scheduling, which consists in minimizing the time elapsed between a due date and the completion date, *i.e.* $|C_j - d_j|$. This elapsed time can also be written $E_j + T_j$. Figure 1.7 p. 27 shows earliness and some of its applications for just-in-time scheduling.



(a) Earliness: $E_j = \max(-L_j, 0)$

(b) Just in time scheduling: $E_j + T_j = |C_j - d_j|$

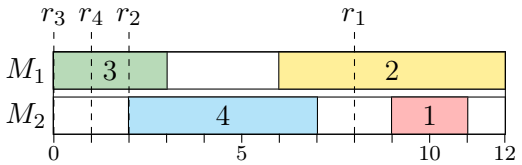
(c) Just in time scheduling with different weights

Figure 1.7: Earliness and its applications for just-in-time scheduling

1.1.5 Regularity and dominance

Except those based on job earliness, all the objective functions defined in section 1.1.4 have an interesting property: the earlier the operations are completed, the lower the costs. This is a property induced by the fact that these objectives are regular [64]. With regular objectives, it is worth scheduling operations as soon as possible, without changing their sequencing on the machines. We hereafter formalize these notions.

Consider, for example, an instance of the $P2|r_j|\gamma$ problem, regardless of the objective function γ. Figure 1.8 shows an admissible schedule S of this problem.



(a) Schedule S on a Gantt chart

$$S = \big((11,2),(12,1),(3,1),(7,2)\big)$$

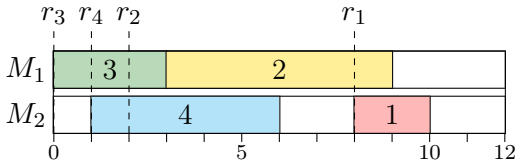
$$S = \Bigg[\underbrace{\begin{pmatrix}(3,2)\\(4,1)\end{pmatrix}}_{\text{jobs}}, \underbrace{\begin{pmatrix}(3,12)\\(7,11)\end{pmatrix}}_{\text{completions}} \Bigg]$$

(b) List representations of S

Figure 1.8: Two representations of a schedule S as a list of values

A schedule S is defined by the placement of all its operations. It is a list $S = ((C_1, I_1), \ldots, (C_n, I_n))$ whose each couple (Completion time, Machine) $= (C_j, I_j)$ forms the coordinates of the end of the segment corresponding to operation O_j on a Gantt chart. Another representation can be used, closer to the Gantt chart: sort the operations by machine and then by chronological order, and store the completion times in another list with the same structure, as shown in figure 1.8b.

An admissible schedule is said to be semi-active when no operations can be scheduled earlier without violating a constraint and without changing the relative ordering of operations (*i.e.* their machines and chronological order). Figure 1.9 shows a semi-active schedule S'.



(a) A semi-active schedule S'

$$S' = \begin{pmatrix}(3,2)\\(4,1)\end{pmatrix}$$

(b) List representations of S'

Figure 1.9: A semi-active schedule and its representation as a list of jobs

A semi-active schedule is defined without ambiguity by the placement of all its operations, *i.e.* by the list $S = (j_{ik})_{i,k}$ where i is the machine number and et k represents the chronological order. Indeed, taking into account precedences and release times, completion times of all operations can be computed step by step following the chronological order.

In an optimization problem that consists in minimizing an objective γ, we denote by $\mathscr{S}$ the set of solutions, and let $\mathscr{D} \subset \mathscr{S}$ be a solution subset. The set $\mathscr{D}$ dominates $\mathscr{S}$ when $\min_{s\in\mathscr{D}} \gamma(s) = \min_{s\in\mathscr{S}} \gamma(s)$, *i.e.* when $\mathscr{D}$ contains an optimal solution. If we identify a set $\mathscr{D}$ dominating $\mathscr{S}$, then we can find an optimal solution by restricting the search to elements of $\mathscr{D}$.

With a regular objective γ, the set of semi-active schedules is dominant. To find an optimal solution, we can restrict to semi-active schedules and use the representation in which a schedule is a list of jobs indexed by machines and chronological order.

1.1.6 Examples

We describe several classic examples, which model distinct settings in scheduling. Some of them are famous because there exists a rule to solve them in polynomial time (the notions of complexity theory we use are recalled in section 1.2). These examples serve as introduction to a global picture of the field of scheduling, but also introduce the reader to the problems dealt with in the remainder of this document.

Example 1.1: Sequencing with interval constraints.

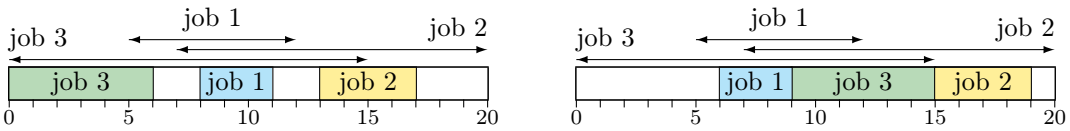


Figure 1.10: Two solutions of an instance of the $1|r_j, \tilde{d}_j|$- problem

This problem is denoted by $1|r_j, \tilde{d}_j|$- and is strongly NP-hard. There is no objective to minimize, and the problem consists in finding an admissible schedule, or more simply a job order which leads to an admissible schedule. The brutal algorithm tries the $n!$ possible orders, but since 1982, the state of the art algorithm is a Branch and Bound procedure due to Carlier [14].

The $1|r_j, \tilde{d}_j|$- and $1|r_j|L_{\max}$ problems are equivalent, *i.e.* there is a polynomial reduction from one to another. The $1|r_j, \tilde{d}_j|$- problem is often considered as an optimization block used to solve many other problems, e.g. shop problems.

Example 1.2: Some classic rules for sequencing problems.

The $1|d_j|L_{\max}$ problem consists in sequencing jobs to minimize the maximum lateness. It is solved to optimality by the Jackson's rule [39], which consists in sequencing jobs in EDD order, *i.e.* Earliest Due Date First.

The $1|d_j|L_{\max}$ problem is a particular case of the $1|prec|f_{\max}$ problem, where f is a regular individual cost. The $1|prec|f_{\max}$ problem consists in sequencing jobs in presence of precedence constraints to minimize a maximum regular cost. This problem is solved to

optimality in polynomial time by Lawler's algorithm [47], or LCL (Lowest Cost Last) rule: from the end to the beginning, schedule the job with a minimal cost among jobs without a successor.

The $1|d_j|\sum U_j$ problem consists in sequencing jobs to minimize the number of tardy jobs. This problem is solved to optimality in polynomial time by Moore's algorithm [55]: schedule jobs in EDD order. Then, each time a job is scheduled tardy, sacrifice the longest of already scheduled jobs by moving it to the end of the schedule.

The interest of these rules goes beyond the specific problems they solve. Many heuristics or lower bounds used in Branch-and-Bound procedures (described in section 1.3.3) exploit these rules. For example:

- In example 1.3, Smith's rule polynomially solves the $1||\sum w_jC_j$ problem to optimality. A straightforward extension of this rule is applied as a heuristic (called Smith's heuristic) for the NP-hard problem $1|\tilde{d}_j|\sum w_jC_j$.
- In example 1.5 p. 30, Johnson's rule polynomially solves the $F2||C_{\max}$ problem to optimality. Many heuristics are derived from this rule to cope with the NP-hard problem $F|prmu|C_{\max}$.
- Moore's algorithm polynomially solves the $1|d_j|\sum U_j$ problem to optimality. Della Croce et al. [20] use Moore's algorithm to analyze the NP-hard problem $P|d_j|\sum U_j$ which appears in example 1.7 p. 31.

Example 1.3: Sequencing to minimize the total weighted completion time.

Let us consider a pier in a harbor on which boats are docked one after the other. The end of the pier is the exit of the harbor. Each boat j has a length p_j, a speed $1/w_j$, a draught that forces it to be at a distance of at most $\tilde{d}_j$ from the exit. How to park boats to minimize the average time to exit the harbor?

This real situation is modeled by the $1|\tilde{d}_j|\sum w_jC_j$ problem. This is a strongly NP-hard sequencing problem, in which a schedule is uniquely defined by the order of jobs. So, the brutal algorithm tries the $n!$ possible job orders. The current state of the art algorithm is due to Shang et al. [81].

The $1|\tilde{d}_j|\sum w_jC_j$ problem has two well-known polynomial sub-problems: the $1||\sum w_jC_j$ and $1|\tilde{d}_j|\sum C_j$ problems. The former is solved to optimality in polynomial time by Smith's rule [82], which consists in sorting jobs in increasing order of p_j/w_j, or WSPT (Weighted Shortest Processing Time first). The latter is solved to optimality in polynomial time by an extension of Smith's rule: from the end to the beginning, among jobs which do not violate their deadline, schedule the one with the largest ratio p_j/w_j.

Example 1.4: Just-in-time scheduling.

This problem is denoted by $1|d_j|\sum E_j+T_j$. There are n jobs j to schedule on one machine. For each job j, there is a due date d_j. This problem models a usual real-life situation, where a job j represents a hard to store product needed at some date d_j. It is obviously a disadvantage if the product is delivered late, so there is a tardiness penalty T_j, but it is also a disadvantage if the product is delivered early, because it is difficult to store, so there is an earliness penalty E_j.

Ideally, we would like that each job j ends exactly at date d_j, which corresponds to the ideal $1|C_j{=}d_j|$- problem, but this problem is trivial to solve and potentially unfeasible: there is only one possible ideal schedule, and it is rarely admissible. We therefore try, as a best approximation of this ideal solution, to minimize the sum of the distances between the nominal end date *i.e.* the due date d_j and the actual end date *i.e.* the completion time C_j. The value of each distance is $|C_j - d_j| = E_j + T_j$. Notice that this objective is not regular, so standard techniques are powerless to solve scheduling problems aimed at minimizing it.

Example 1.5: permutation flowshop to minimize makespan.

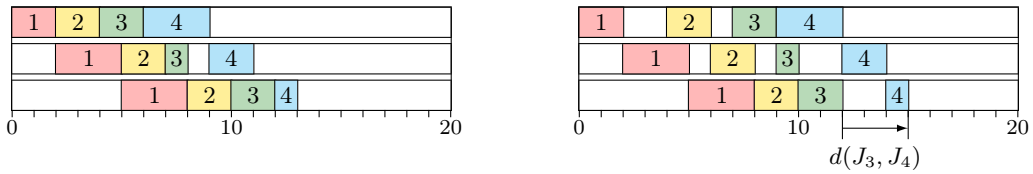
This problem, denoted by $F|prmu|C_{\max}$, models for example an assembly line for agricultural machinery. All the devices share a common platform but have a multitude of options depending on their intended use. It is therefore necessary to produce small series, established in advance by the order book. In the assembly line, all the devices go, in the order of the conveyor and without overtaking each other, through the same assembly stations, but with different processing times depending on the options.

It turns out that for this objective and for 2 and 3 machines, the optimal solutions of the permutation flowshop are also optimal for the non-permutation flowshop. In other words the $F2|prmu|C_{\max}$ and $F2||C_{\max}$ problems are equivalent, and the $F3|prmu|C_{\max}$ and $F3||C_{\max}$ problems are equivalent.

For 2 machines, the $F2||C_{\max}$ problem is solved to optimality in polynomial time by Johnson's rule [41] also called SPT(1)–LPT(2) rule: schedule at the beginning the jobs whose 1^{st} operation is shorter than the 2^{nd} one, and sort them by shortest processing time of the 1^{st} operation first; schedule at the end the jobs whose 1^{st} operation is longer than the 2^{nd} one, and sort them by longest processing time of the 2^{nd} operation first.

The $F3||C_{\max}$ problem is strongly NP-hard and is one of the reference problems in scheduling, and even in Operations Research.

The $F|prmu, nowait|C_{\max}$ problem is the version where there must be no delays between two consecutive operations of a same job. It models situations such as the cold chain or conversely steel or glass factories, the treatment of unstable chemical or biological products. It also models a maintenance chain of devices whose capital cost is preponderant (e.g. planes). Figure 1.11 shows a comparison between semi-active schedules with the same job ordering, for the same instance of $F3||C_{\max}$ and $F3|prmu, nowait|C_{\max}$ problems.



(a) $F3||C_{\max}$ problem (b) $F3|nowait|C_{\max}$ problem

Figure 1.11: Same job ordering for the $F3||C_{\max}$ and $F3|nowait|C_{\max}$ problems

The $F|prmu, nowait|C_{\max}$ problem reduces to the Asymmetric Traveling Salesman Problem. The distance between two consecutive jobs j and j' is the difference $(C_{j'} - C_j)$ between their completion times, which only depends on the processing times of j and j'.

Example 1.6: Bin Packing.

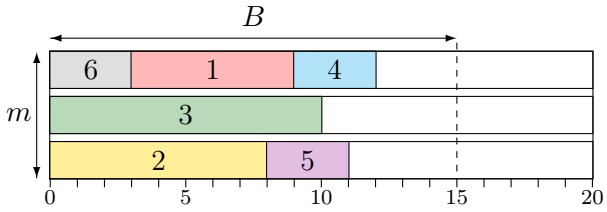


Figure 1.12: Correspondence between the $P||C_{\max}$ problem and bin packing

The bin packing problem can be stated as follows: given bins, all with capacity B, and n objects of sizes $p_1, \ldots, p_n$ how many bins are needed to store the objects in the bins (and how to distribute them)?

There is a direct correspondence between the bin packing problem and the makespan minimization problem with identical parallel machines: an object corresponds to a job and a bin corresponds to a machine, as shown on Figure 1.12.

The $P||C_{\max}$ problem and the bin packing problem both reduce to the same decision problem $\mathcal{P}_{dec}$: we define $\mathcal{P}_{dec}(m, B)$ as true if and only if there exists a distribution into m bins with capacity B, or if and only if there exists an admissible schedule on m machines and with makespan at most B.

Bin packing consists in finding $\min\{m \mid \mathcal{P}_{dec}(m, B)\}$ for a fixed B. The $P||C_{\max}$ problem consists in finding $\min\{B \mid \mathcal{P}_{dec}(m, B)\}$, for a fixed m. The decision problem $\mathcal{P}_{dec}$, and therefore the $P||C_{\max}$ and bin packing problems are strongly NP-hard (in the general case where m and B are unbounded).

Example 1.7: Parallel machine scheduling to minimize penalties.

A factory, with several identical production lines must deliver to clients, and each client has been promised a due delivery date d_j. We try to minimize the penalties for delay. All these problems are modeled by a problem of type $P||\sum f_j$. Table 1.13 details them according to the penalty mode (flat-rate or daily, same for all clients or on a per client basis). These four problems are NP-hard.

Table 1.13: Modeling of penalties

penalties are	flat-rate	daily
same for all	$P\|\|\sum U_j$	$P\|\|\sum T_j$
per client	$P\|\|\sum w_j U_j$	$P\|\|\sum w_j T_j$

1.2 Complexity of algorithms and problems

The theory of computational complexity is a very vast domain. We recall basic notions, as defined e.g. in Garey and Johnson [27], and we adapt the notations to the domain of scheduling.

A problem $\mathcal{P}$ can be specified as a function $\mathcal{P} : D \longrightarrow R$, which takes as parameter a data, or instance $\mathcal{I} \in D$, and which computes a result $\mathcal{P}(\mathcal{I}) \in R$. An algorithm $\mathcal{A}$ solving $\mathcal{P}$ is a concrete list of instructions implementing the function $\mathcal{P}$ on a machine.

A problem can be solved by many different algorithms, with many different performances. The (computational) complexity of an algorithm translates its consumption in resources (time, space) when the instance varies.

We recall well-known notions, while precisely defining the notations. Section 1.2.1 deals with asymptotic comparison and Landau's notations. Section 1.2.2 deals with the size and measure of an instance. Section 1.2.4 deals with the worst-case complexity of algorithms and problems, the main complexity classes, and the notion of polynomial reduction. Finally, section 1.2.6 deals with polynomial reductions between scheduling problems.

1.2.1 Landau notations

Landau's notations enable to compare asymptotically expressions of a variable when the variable becomes large. There are many variants of this notation, so it is necessary to define precisely the one we use in the context of the complexity study.

We use the very standard O (upper bound) and Θ (tight upper bound) notations, and we express them using the limit of a supremum ($\limsup$). We do not use the notation Ω, whose definition for algorithmics is controversial (according to some authors, $f = \Omega(g)$ means that f is not negligible compared to g, i.e $f \neq o(g)$; according to some others, it means $g = O(f)$).

In its usual definition, the Landau notation O refers to a variable, say x, real or integer, and asymptotically compares two real expressions $f(x)$ and $g(x)$. It states that $f(x)$ is bounded up to a multiplicative constant by $g(x)$ when x becomes large. We usually define:

$$f(x) = \underset{x\to+\infty}{O}(g(x)) \iff \exists U > 0,\ \exists x_0,\ \forall x \geqslant x_0, |f(x)| \leqslant U|g(x)| \tag{1.2}$$

In a context where the functions to be compared $f(x)$ and $g(x)$ are always (strictly) positive, which will always be the case for us, this definition is equivalent to studying $f(x)/g(x)$. By symmetry, we write the constant U as the ratio of two constants $U = U_f/U_g$. This symmetrical writing will be useful later. We have:

$$\begin{aligned} f(x) = \underset{x\to+\infty}{O}(g(x)) &\iff \exists U_f > 0, U_g > 0,\ \exists x_0,\ \forall x \geqslant x_0,\ \frac{f(x)}{g(x)} \leqslant \frac{U_f}{U_g} \\ &\iff \exists U_f > 0, U_g > 0,\ \exists x_0,\ \forall x \geqslant x_0,\ \frac{f(x)}{U_f}\frac{U_g}{g(x)} \leqslant 1 \\ &\iff \exists U_f > 0, U_g > 0,\ \exists x_0,\ \sup_{x\geqslant x_0} \frac{f(x)}{U_f}\frac{U_g}{g(x)} \leqslant 1 \end{aligned} \tag{1.3}$$

We will adapt these definitions in the case of worst-case complexities. We assume a predefined function $|\cdot|$ which, to each instance $\mathcal{I}$, associates a size $|\mathcal{I}|$. This size intuitively represents the data volume of the instance, but it does not need to be precisely defined now. We only require that there exist instances of arbitrarily large sizes: $\forall x_0, \exists \mathcal{I}$, $|\mathcal{I}| \geqslant x_0$. In other words, $\{\mathcal{I} \mid |\mathcal{I}| \geqslant x_0\}$ is never empty.

We group the instances by equal sizes and compare expressions $f(\mathcal{I})>0$ and $g(\mathcal{I})>0$, representing complexities, when the size becomes large. In the case of complexities, smaller is better, so the worst case corresponds to studying, for a given size x, the largest ratio among instances of this size, *i.e.* $\sup_{|\mathcal{I}|=x} f(\mathcal{I})/g(\mathcal{I})$. Precisely, this supremum combines nicely with the supremum of the formula (1.3) p. 32. We define the notation $f(\mathcal{I}) = O_{|\mathcal{I}|\to+\infty}(g(\mathcal{I}))$ meaning that g is a worst case (upper) bound of f:

$$f(\mathcal{I}) = \underset{|\mathcal{I}|\to+\infty}{O}(g(\mathcal{I})) \iff \exists\, U_f > 0, U_g > 0,\ \exists\, x_0,\ \sup_{|\mathcal{I}|\geqslant x_0} \frac{f(\mathcal{I})}{U_f}\frac{U_g}{g(\mathcal{I})} \leqslant 1 \tag{1.4}$$

On the same principle, we also define the notation $f(\mathcal{I}) = \Theta_{|\mathcal{I}|\to+\infty}(g(\mathcal{I}))$ meaning that g is a worst case tight bound of f:

$$f(\mathcal{I}) = \underset{|\mathcal{I}|\to+\infty}{\Theta}(g(\mathcal{I})) \iff \begin{cases} \exists\, U_f > 0, U_g > 0,\ \exists\, x_0,\ \sup\limits_{|\mathcal{I}|\geqslant x_0} \dfrac{f(\mathcal{I})}{U_f}\dfrac{U_g}{g(\mathcal{I})} \leqslant 1 \\ \exists\, L_f > 0, L_g > 0,\ \exists\, x_0,\ \sup\limits_{|\mathcal{I}|\geqslant x_0} \dfrac{f(\mathcal{I})}{L_f}\dfrac{L_g}{g(\mathcal{I})} \geqslant 1 \end{cases} \tag{1.5}$$

In practice, g is an (upper) bound for f when, up to a multiplicative constant, $f(\mathcal{I}) \leqslant g(\mathcal{I})$ for all sufficiently large instances. The bound is tight when, in addition, there are instances of arbitrarily large sizes for which $f(\mathcal{I}) \simeq g(\mathcal{I})$.

The traditional O and Θ notations mean "bounded up to a constant". The star-notations O^* and Θ^*, introduced by Woeginger [90], mean "bounded up to a polynomial in the size of the instance". They are very useful to simplify the calculations in the case of exponential bounds. The definitions are transposed by replacing the constants by polynomials (with positive values for a sufficiently large instance size):

$$f(\mathcal{I}) = \underset{|\mathcal{I}|\to+\infty}{O^*}(g(\mathcal{I})) \iff \exists\, P_f, P_g : \text{poly},\ \exists\, x_0,\ \sup_{|\mathcal{I}|\geqslant x_0} \frac{f(\mathcal{I})}{P_f(x)}\frac{P_g(x)}{g(\mathcal{I})} \leqslant 1 \tag{1.6}$$

$$f(\mathcal{I}) = \underset{|\mathcal{I}|\to+\infty}{\Theta^*}(g(\mathcal{I})) \iff \begin{cases} \exists\, P_f, P_g : \text{poly},\ \exists\, x_0,\ \sup\limits_{|\mathcal{I}|\geqslant x_0} \dfrac{f(\mathcal{I})}{P_f(x)}\dfrac{P_g(x)}{g(\mathcal{I})} \leqslant 1 \\ \exists\, Q_f, Q_g : \text{poly},\ \exists\, x_0,\ \sup\limits_{|\mathcal{I}|\geqslant x_0} \dfrac{f(\mathcal{I})}{Q_f(x)}\dfrac{Q_g(x)}{g(\mathcal{I})} \geqslant 1 \end{cases} \tag{1.7}$$

It is very common to abbreviate the symbol $\mathcal{I}$ and the indication of limit in the notations. For example, the phrase $t = O^*(2^n)$ actually means $t(\mathcal{I}) = O^*{}_{|\mathcal{I}|\to+\infty}\left(2^{n(\mathcal{I})}\right)$.

1.2.2 Size of an instance

An instance must be encoded as a word made up of symbols belonging to an alphabet. This encoding is purely conventional, as long as it is "reasonable", a term that we will specify later. Consider a simple but representative example, where an instance can be seen as a list of values, say natural numbers: $\mathcal{I} = (v_1, \ldots, v_\ell)$. It is agreed that the alphabet contains as symbols the parentheses, the comma and the digits, and we choose to encode the instance by copying exactly the mathematical notation of the list. For example, the instance $\mathcal{I} = (5, 1, 8)$ is encoded as "`(5,1,8)`".

We define the size $|\mathcal{I}|$ of the instance $\mathcal{I}$ as asymptotically proportional to the number of symbols used to encode the instance, definition very well suited to the Landau's O notation. For our representative example $\mathcal{I} = (v_1, \ldots, v_\ell)$, we take $|\mathcal{I}| = \ell + \sum_i \log v_i$. We

can now clarify the notion of "reasonable" encoding: it is an encoding where the number of symbols depends only logarithmically on the instance values. Specifically, this involves coding them, for example, in binary or decimal, but not in unary.

In the complexity calculations, we consider that the size $|\mathcal{I}|$ goes to $+\infty$, but the expression $|\mathcal{I}|$ rarely appears explicitly in calculations, because it suffers from two disadvantages. On the one hand $|\mathcal{I}|$ depends (logarithmically) on the data value because of the term $(\sum_i \log v_i)$. Due to this, in the calculations it is much easier to use the number of instance values (the length ℓ) rather than the size $|\mathcal{I}|$ itself. On the other hand, the number of instance values depends a lot on the encoding. For example, a graph with n vertices is encoded by n^2 values with an adjacency matrix, but potentially much less with the list of edge ends.

To avoid these disadvantages, we choose by convention parameters, called size parameters, which characterize, depending on the problem, the fundamental number of data of an instance regardless of its encoding, and it is used to express the complexities. Thus, it is necessary to choose once for all a reasonable encoding, and then focus on the essential: these parameters. Traditionally, the main size parameter is systematically called n, and we can introduce others. For example, in the case of a graph, the main size parameter is the number n of vertices, and it is common to use as 2^{nd} size parameter the number m of edges.

1.2.3 Measure of an instance

As we have seen, the size of an instance is asymptotically proportional to the number of symbols used to encode it. This notion is valid only if the instance is encoded in a reasonably compact way. In particular, the integers that appear in the instance must be encoded in binary (or any other base $b \geqslant 2$).

On the same principle, the measure of an instance is asymptotically proportional to the number of symbols used to encode it when numbers are written in unary, *i.e.* when each number is written with as many groups of symbols than its value. For example, the instance $\mathcal{I} = (5, 1, 8)$ can be encoded in unary as "`(0+1+1+1+1+1, 0+1, 0+1+1+1+1+1+1+1+1)`". The measure of an instance is therefore its number of symbols in a deliberately unreasonable encoding.

It may seem artificial to write integers in unary. Yet this is what we do without realizing it, each time we list the elements of a multiset one by one instead of listing them with their order of multiplicity. For example, writing $80 = 2 \times 2 \times 2 \times 2 \times 5$ instead of $80 = 2^4 \times 5$ is equivalent to writing the exponent 4 in unary. Similarly, writing a polynomial in the form $P = (X - r_1) \cdots (X - r_n)$ instead of $P = (X - z_1)^{m_1} \cdots (X - z_k)^{m_k}$ is equivalent to writing all the multiplicities m_i in unary.

The notion of measure is used to capture this kind of situation. As well as the size of an instance is essentially defined as $|\mathcal{I}| = \ell + \sum_i \log v_i$, likewise the measure of an instance is essentially defined as $||\mathcal{I}|| = \ell + \sum_i v_i$. We use the measure of an instance in a context where the sum of the values is preponderant compared to their number, we then have $||\mathcal{I}|| = \Theta(\sum_i v_i)$. Note that the maximum and the sum are polynomially related, so in the end, for an instance $\mathcal{I} = (v_1, \ldots, v_\ell)$, we have $||\mathcal{I}|| = \Theta(\sum_i v_i) = \Theta^*(\max_i v_i)$. We also have that $\log ||\mathcal{I}||$ is polynomial in $|\mathcal{I}|$, *i.e.* $\log ||\mathcal{I}|| = O^*(1)$.

1.2.4 Worst case complexity

The two main resources used by an algorithm are time and space (memory). We describe time complexity, but the principle is the same for space complexity. Given an algorithm $\mathcal{A}$ and an instance $\mathcal{I}$, we define $t_{\mathcal{A}}(\mathcal{I})$ as the time, expressed in an arbitrary unit, e.g. the number of steps on an abstract Turing machine, used by the algorithm $\mathcal{A}$ to achieve its computation on instance $\mathcal{I}$. Therefore, this defines a function $t_{\mathcal{A}} : D \longrightarrow]0, +\infty[$. Complexity is concerned with asymptotic behavior of the function $t_{\mathcal{A}}$ when the size of the instance becomes large, and consists in comparing the function $t_{\mathcal{A}}$ to a reference function $f : D \longrightarrow]0, +\infty[$. For this, we use the Landau notations and we are looking to write formulas of the form $t_{\mathcal{A}} = O(f)$ or similar.

The (intrinsic) complexity of a problem is, by definition, the lowest complexity (the infimum) which it is possible to reach for an algorithm that solves it. As with algorithms, we focus on worst-case complexities in time and space. For the purpose of definitions, we restrict ourselves to decision problems, *i.e.* to problems $\mathcal{P}_{dec} : D \longrightarrow \mathbb{B}$ whose answer is a boolean, "yes" or "no". An instance $\mathcal{I}$ such that $\mathcal{P}_{dec}(\mathcal{I}) =$ "yes" is by definition a "yes-instance" of the problem $\mathcal{P}_{dec}$.

The complexity class $\mathcal{P}$ contains the decision problems of worst case time complexity at most polynomial, *i.e.* in $O^*(1)$. Problems outside $\mathcal{P}$ therefore have super-polynomial complexities.

The class $\mathcal{NP}$ is defined as follows: a verifier of problem $\mathcal{P}_{dec}$ is a decision problem $\mathcal{V} : D \times C \longrightarrow \mathbb{B}$ which takes as parameter an instance and a "certificate", such that $\forall \mathcal{I} \in D,\ \mathcal{P}_{dec}(\mathcal{I}) \iff \exists\, c \in C \mid \mathcal{V}(\mathcal{I}, c)$. Class $\mathcal{NP}$ contains the decision problems that admit a polynomial worst-case time checker with respect to $|\mathcal{I}|$.

The complexity class *EXPTIME* contains the decision problems whose worst-case time complexity is bounded by the exponential of a polynomial, therefore in $O^*(2^{n^d})$ for a certain degree d. This is the case for complexities in $O^*(1)$, $O^*(\alpha^n)$ with $\alpha > 1$, $O^*(n!)$, $O^*(n^n)$, $O^*(2^{n^2})$ but not $O^*(2^{2^n})$.

We now define the notion of polynomial reduction between two decision problems: a problem $\mathcal{P}_{dec} : D \longrightarrow \mathbb{B}$ reduces polynomially to a problem $\mathcal{P}'_{dec} : D' \longrightarrow \mathbb{B}$ if there is a function $f : D \longrightarrow D'$ computable in polynomial time, which transforms an instance of $\mathcal{P}_{dec}$ into an equivalent instance of $\mathcal{P}'_{dec}$, *i.e.* $\forall \mathcal{I}, \mathcal{P}_{dec}(\mathcal{I}) \iff \mathcal{P}'_{dec}(f(\mathcal{I}))$. In this case we write $\mathcal{P}_{dec} \propto \mathcal{P}'_{dec}$, and we say that $\mathcal{P}'_{dec}$ is at least as difficult as $\mathcal{P}_{dec}$.

A problem is said to be NP-hard when it is at least as hard as any problem in $\mathcal{NP}$. A NP-complete problem is a problem which is both in $\mathcal{NP}$ and NP-hard. We have $\mathcal{P} \subset \mathcal{NP} \subset EXPTIME$ and the famous conjecture $\mathcal{P} \neq \mathcal{NP}$ asserts that the inclusion is strict. Equivalently, it asserts that no NP-complete problem can be solved in polynomial time. This conjecture is widely accepted.

Among the NP-hard problems, we make a new distinction: problems which are polynomial with respect to the instance measure $||\mathcal{I}||$ are said to be pseudopolynomial or weakly NP-hard; problems which remain NP-hard under these conditions are said to be strongly NP-hard. Finally, we can classify decision problems in increasing order of theoretical difficulty: polynomial, weakly $\mathcal{NP}$, strongly $\mathcal{NP}$, exponential. Figure 1.14 p. 36 illustrates the nesting of complexity classes, assuming that $\mathcal{P} \neq \mathcal{NP}$.

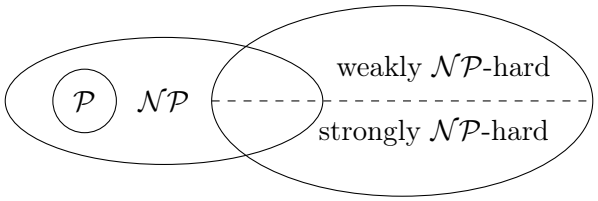


Figure 1.14: Venn Diagram of some basic complexity classes, assuming $\mathcal{P} \neq \mathcal{NP}$

All these definitions extend to general problems, because the solution of a general problem can be reduced, under reasonable assumptions, to the solution of a polynomial number of decision problems. As an example, consider a generic problem $\mathcal{P}_{gen} : D \longrightarrow \mathbb{B}^*$ whose results are expressed in the form of a sequence of booleans. Suppose the length of the result is polynomial in the instance size: $|\mathcal{P}_{gen}(\mathcal{I})| = O^*(1)$. We define this decision problem $\mathcal{P}_{dec} : D \times \mathbb{B}^* \longrightarrow \mathbb{B}$ by: for any instance $\mathcal{I}$, for any sequence π, $\mathcal{P}_{dec}(\mathcal{I}, \pi)$ tests whether π is a prefix of the result $\mathcal{P}_{gen}(\mathcal{I})$, *i.e.* $\mathcal{P}_{dec}(\mathcal{I}, \pi) \Longleftrightarrow \exists\, \sigma \in \mathbb{B}^* \mid \pi \cdot \sigma = \mathcal{P}_{gen}(\mathcal{I})$. By trying longer and longer prefixes, $\mathcal{P}_{gen}(\mathcal{I})$ is determined in at most $2|\mathcal{P}_{gen}(\mathcal{I})|$ steps.

1.2.5 Application to scheduling problems

In scheduling, the size of the instance is systematically the number of jobs n, and possibly the number of machines m. So, the symbol n defined in section 1.1.1 as the number of jobs coincides with the symbol n as the instance size. We take as measure of the instance the sum of the values of the instance: processing times, release dates, due dates or deadlines, weight of individual costs of the objective function.

Scheduling problems are optimization problems, which consist in minimizing an objective function. In some cases, the function to be minimized is part of the instance, and we have to encode this function in a reasonable way. We explain the issue hereafter:

Let $\gamma : \mathbb{N} \longrightarrow \mathbb{N}$ be an objective function to encode. We must avoid encoding γ in an extensional way, for example by a list $(\gamma(0), \gamma(1), \gamma(2), \ldots, \gamma(C_{\max}))$ in the case of a single-machine problem. Indeed, the number of values to be stored is of the order of magnitude of the sum of processing times $\sum p_j$. This encoding is unreasonable: it is of pseudopolynomial length instead of polynomial length and therefore its length is potentially exponential in n.

On the contrary, γ must be encoded by its calculation formula, mostly of constant size, except for the weights w_j whose list is of size n. The computation time of γ is therefore part of the computation time of the algorithm and, as long as scheduling is concerned, can be achieved in polynomial time.

It is convenient to add another assumption, on the value of γ itself: we will systematically assume that the value of the optimum objective γ^{opt} is pseudopolynomial, *i.e.* $\gamma^{opt} = O^*(||\mathcal{I}||^d)$ for some degree d fixed once for all.

These assumptions are not restrictive. They are trivially verified by all the usual cost functions. Each classic individual cost function $f_j \in \{C_j, L_j, T_j, U_j, E_j\}$ is calculated in constant time. Their maximum or sum is calculated in time $O(n)$. The value of each classic individual cost f_i is bounded by the maximum completion time of a schedule, itself in $O^*(||\mathcal{I}||)$. The value of their sum or maximum, therefore the value of γ^{opt}, is also in $O^*(||\mathcal{I}||)$. For weighted versions, we have to multiply by the weights, in $O^*(||\mathcal{I}||)$, which gives a γ^{opt} in $O^*(||\mathcal{I}||^2)$.

1.2.6 Reductions among scheduling problems

As we have seen in section 1.2.4, we write $A \propto B$ to mean that the problem A reduces polynomially to B. We also write $A \propto_0 B$ to refer to the specific case when A is a simple subproblem of B, *i.e.* any instance of A is also an instance of B. In this case, there is a trivial polynomial reduction from A to B.

There are many elementary reductions between scheduling problems, and the advantage of Graham's notation is to make them easily readable: they apply field by field. We describe the simplest ones:

Environments: The single-machine environment is a particular case of all other environments, we have $1 \propto_0 P \propto_0 R$, $1 \propto_0 F|prmu$, $1 \propto_0 F \propto_0 J$.

Constraints: For each constraint set β, we have $\beta \propto_0 \beta, chains \propto_0 \beta, prec$ (consider precedence graphs), $\beta \propto_0 \beta, r_j$ (take $r_j = 0$), $\beta \propto_0 \beta, \tilde{d}_j$ (take $\tilde{d}_j = +\infty$ or a formula of the kind $\tilde{d}_j = \max_{j'} r_{j'} + \sum_{j'} p_{j'}$ which bounds the makespan of any semi-active schedule).

Objectives: Sums reduce to their weighted version: $(\sum_j f_j) \propto_0 (\sum_j w_j f_j)$. Completion time reduces to lateness and tardiness, by taking $d_j = 0$, *i.e.* $C_j \propto_0 L_j$ and $C_j \propto_0 T_j$. Proposition 1.8 hereafter introduces and proves two non trivial reductions, cited in Pinedo [64] and Brucker [12]: $L_{\max} \propto \sum_j U_j$ and $L_{\max} \propto \sum_j T_j$. Figure 1.15 shows the reduction graph between usual objectives:

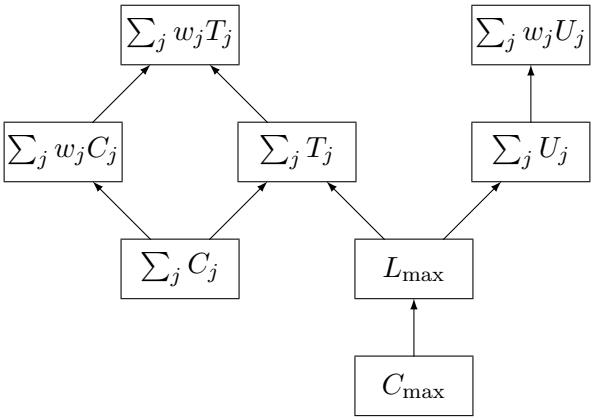


Figure 1.15: Polynomial reductions between usual objectives.

Proposition 1.8. For each environment α and constraint set β, $\alpha|\beta|L_{\max} \propto \alpha|\beta|\sum_j U_j$ and $\alpha|\beta|L_{\max} \propto \alpha|\beta|\sum_j T_j$.

Proof.
Let us consider an instance $\mathcal{I}$ of the $\alpha|\beta|L_{\max}$ problem. This instance contains due dates $(d_1, \ldots, d_n)$. We denote by $\mathcal{I}(d'_1, \ldots, d'_n)$ the same instance in which due dates are $(d'_1, \ldots, d'_n)$. It is not only an instance of the $\alpha|\beta|L_{\max}$ problem but also of the $\alpha|\beta|\sum_j U_j$ and $\alpha|\beta|\sum_j T_j$ problems. All instances $\mathcal{I}(d'_1, \ldots, d'_n)$ share the same admissible schedules independently of the due dates and objectives.

Let $z \in \mathbb{N}$ be a due date shift and S an admissible schedule for $\mathcal{I}$. We have:

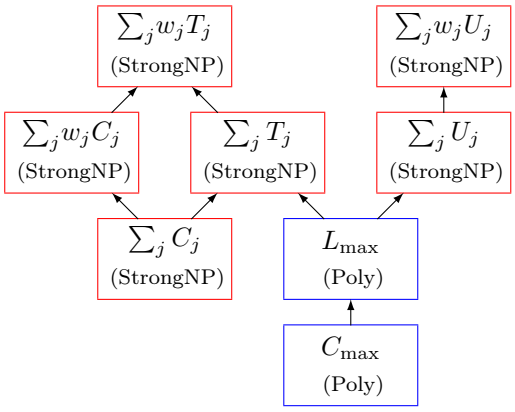
$$\begin{aligned} L_{\max}(S) \leqslant z \text{ for } \mathcal{I}(d_1, \ldots, d_n) &\iff \forall\, j,\ C_j(S) \leqslant d_j + z \\ &\iff \forall\, j,\ U_j(S) = 0 \text{ for } \mathcal{I}(d_1{+}z, \ldots, d_n{+}z) \\ &\iff \textstyle\sum_j U_j(S) = 0 \text{ for } \mathcal{I}(d_1{+}z, \ldots, d_n{+}z) \end{aligned}$$

Taking the optimum, *i.e.* minimum values, we get:

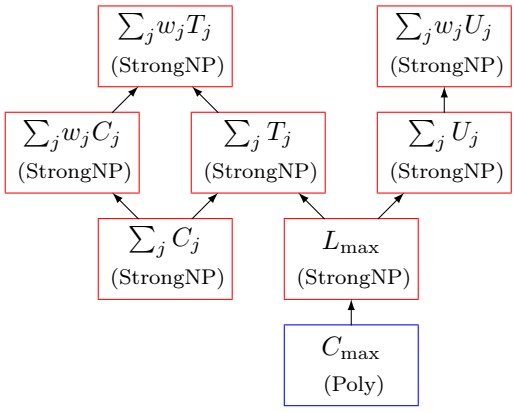
$$\begin{aligned} L_{\max}^{opt} &= \min\{z \in \mathbb{N} \mid \exists\, S \text{ admissible} \mid L_{\max}(S) \leqslant z \text{ for } \mathcal{I}(d_1, \ldots, d_n)\} \\ &= \min\{z \in \mathbb{N} \mid \exists\, S \text{ admissible} \mid \textstyle\sum_j U_j(S) = 0 \text{ for } \mathcal{I}(d_1{+}z, \ldots, d_n{+}z)\} \\ &= \min\{z \in \mathbb{N} \mid \textstyle\left(\sum_j U_j\right)^{opt} = 0 \text{ for } \mathcal{I}(d_1{+}z, \ldots, d_n{+}z)\} \end{aligned} \tag{1.8}$$

So, given an algorithm to solve the $\alpha|\beta|\sum_j U_j$ problem, here is an algorithm to solve the $\alpha|\beta|L_{\max}$ problem: determine by a binary search the minimum z of equation (1.8). An optimal schedule of $\alpha|\beta|\sum_j U_j$ for $\mathcal{I}(d_1{+}z, \ldots, d_n{+}z)$ is also an optimal schedule of $\alpha|\beta|L_{\max}$ for $\mathcal{I}$. The binary search takes $O(\log L_{\max}) = O(\log ||\mathcal{I}||) = O^*(1)$ steps, hence the polynomial reduction. The same rationale holds for $\sum_j T_j$, because $\sum_j T_j{=}0 \Leftrightarrow \sum_j U_j{=}0$. □

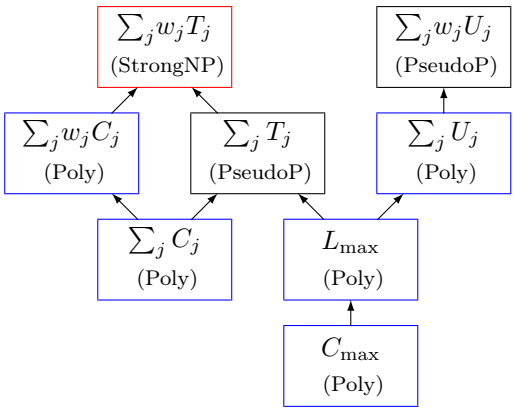
Figures 1.16 and 1.17 p. 39 show the reduction graphs for each class of problems $\alpha \in \{1, P\}$ and for each constraint set $\beta \subset \{r_j, prec\}$. For each problem it is also indicated if it is polynomial (Poly), pseudopolynomial (PseudoP) or strongly NP-hard (StrongNP).



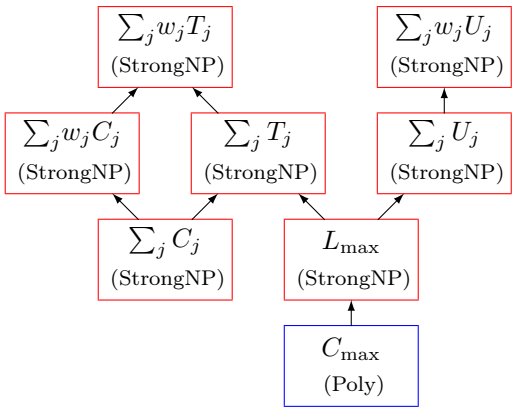
(a) Reductions for $1|prec|\gamma$



(b) Reductions for $1|prec, r_j|\gamma$

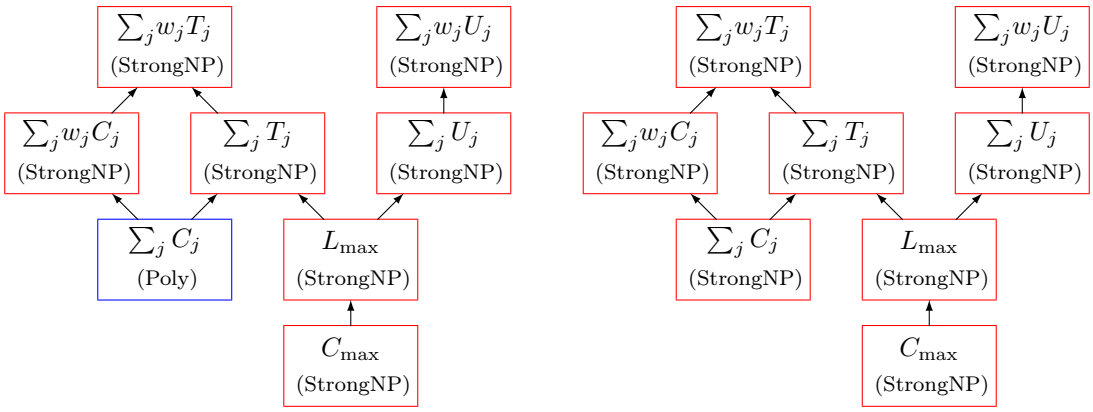


(c) Reductions for $1||\gamma$



(d) Reductions for $1|r_j|\gamma$

Figure 1.16: Reduction graphs and complexity classes of single-machine problems



(a) Reductions for $P||\gamma$

(b) Reductions for $P|r_j$ or $prec|\gamma$

Figure 1.17: Reduction graphs and complexity classes of parallel-machine problems

1.3 Exact Exponential algorithms

We describe state-of-the-art exact algorithms for scheduling. We focus on deterministic, exact, exponential time algorithms, with a proved upper bound on worst case complexities. So, there are no heuristics, nor probabilistic or stochastic algorithms. Exact Exponential algorithms can be divided into four main classes: dynamic programming (section 1.3.1), Sort and Search (section 1.3.2), branching (section 1.3.3), and Inclusion-Exclusion (section 1.3.4, further developed in chapter 2).

1.3.1 Dynamic Programming

Generally speaking, dynamic programming consists in computing a function defined by recursive equations, and remembering intermediate computations, called states. Dynamic programming across the subsets of a job set is well suited for permutation scheduling problems, including single-machine and flowshop problems. But it is very memory consuming, because the number of subsets, thus the number of states, grows exponentially with the number of jobs.

Table 1.18 p. 40 shows some scheduling problems solved using this technique. A direct application of this technique is described by Fomin and Kratsch [23]. Shang et al. [80] apply it to the flowshop problem, and bound the number of states thanks to a fine analysis of the number of critical paths in a schedule. Lenté et al. [50] use a tricky version of dynamic programming across the subsets, applying it to a variable set of 2^k machines together. Cygan et al. [18] also use dynamic programming across the subsets, but they modify the recurrence equations to avoid some subsets which are guaranteed not to lead to an optimal solution. They also apply an appropriate treatment to certain special cases. Then, they count the subsets explored by the dynamic programming procedure and show that their algorithm runs in time $O^*(2-\varepsilon)^n$, with $\varepsilon > 0$ but small ($\varepsilon \simeq 5 \cdot 10^{-16}$ in their proof).

Table 1.18: Time and space worst-case complexity bounds of some scheduling problems solved by Dynamic Programming

problem	reference	time	space
$1\|\|f_{\max}$	Fomin and Kratsch [23]	$O^*(2^n)$	$O^*(2^n)$
$1\|\|\sum_j f_j$	Fomin and Kratsch [23]	$O^*(2^n)$	$O^*(2^n)$
$F3\|prmu\|C_{\max}$	Shang et al. [80]	$O^*(3^n)$	$O^*(3^n)$
$F3\|prmu\|C_{\max}$	Shang et al. [80]	$O^*(2^n\|\|\mathcal{I}\|\|)$	$O^*(2^n\|\|\mathcal{I}\|\|)$
$F3\|prmu\|f_{\max}$	Shang et al. [80]	$O^*(5^n)$	$O^*(5^n)$
$F3\|prmu\|f_{\max}$	Shang et al. [80]	$O^*(2^n\|\|\mathcal{I}\|\|^2)$	$O^*(2^n\|\|\mathcal{I}\|\|^2)$
$F3\|prmu\|\sum_j f_j$	Shang et al. [80]	$O^*(5^n)$	$O^*(5^n)$
$F3\|prmu\|\sum_j f_j$	Shang et al. [80]	$O^*(2^n\|\|\mathcal{I}\|\|^2)$	$O^*(2^n\|\|\mathcal{I}\|\|^2)$
$P\|\|f_{\max}$	Lenté et al. [50]	$O^*(3^n)$	$O^*(2^n)$
$P\|\|\sum_j f_j$	Lenté et al. [50]	$O^*(3^n)$	$O^*(2^n)$
$F2\|\|C^k_{\max}$	Lenté et al. [50]	$O^*(\sqrt{2}^n)$	$O^*(\sqrt{2}^n)$
$1\|prec\|\sum_j C_j$	Cygan et al. [18]	$O^*((2-\varepsilon)^n)$	$O^*((2-\varepsilon)^n)$

1.3.2 Sort and Search

We can separate searching techniques into two main families: global and local. Local search essentially consists in restricting the search for an optimum to the neighborhood of an initial solution. Local search is very valuable to derive efficient heuristics, but it usually offers no guarantee of optimality. We now focus on global searching techniques.

The Sort and Search technique has been introduced by Horowitz and Sahni [36]. It consists in viewing an $O^*(2^{\alpha n})$ sized search space S as the product of two $O^*(2^{\alpha n/2})$ sub-spaces $S = A \times B$ related by a function f such that solutions (a, b) verify $f(a) = b$. By sorting B and searching in B each $f(a), a \in A$, time complexity can be shrunk from $O^*(2^{\alpha n})$ to $O^*(2^{\alpha n/2})$. Table 1.19 shows some scheduling problems solved using this technique.

Table 1.19: Time and space worst-case complexity bounds of some scheduling problems solved by Sort and Search

problem	reference	time	space
$1\|d_j\|\sum_j w_j U_j$	Lenté et al. [49]	$O^*(\sqrt{2}^n)$	$O^*(\sqrt{2}^n)$
$P2\|\|C_{\max}$	Lenté et al. [49]	$O^*(\sqrt{2}^n)$	$O^*(\sqrt{2}^n)$
$P3\|\|C_{\max}$	Lenté et al. [49]	$O^*(\sqrt{3}^n)$	$O^*(\sqrt{3}^n)$
$P\|d_j\|\sum_j w_j U_j$	Lenté et al. [49]	$O^*((m+1)^{\frac{n}{2}}(\frac{n}{2})^{m+2})$	$O^*((m+1)^{\frac{n}{2}}(\frac{n}{2})^{m+2})$
$P2\|d_j\|\sum_j w_j U_j$	Lenté et al. [49]	$O^*(\sqrt{3}^n)$	$O^*(\sqrt{3}^n)$
$P\|\|C_{\max}$	Lenté et al. [49]	$O^*(m^{\frac{n}{2}}(\frac{n}{2})^{m+1})$	$O^*(m^{\frac{n}{2}}(\frac{n}{2})^{m+1})$
$1\,\|d_j = d \geqslant \sum_j p_j$ $\|\sum_j w_j(E_j+T_j)$	T'kindt et al. [86]	$O^*(\sqrt{2}^n)$	$O^*(\sqrt{2}^n)$

Continued on next page

Table 1.19 – continued from previous page

problem	reference	time	space
$P\|d_j = d \geqslant \sum_j p_j \mid \sum_j w_j(E_j+T_j)$	T'kindt et al. [86]	$O^*(3^n)$	$O^*((1+\sqrt{2})^n)$
$P\|d_j\|C_{\max}, L_{\max}$	Shang and T'Kindt [79]	$O^*(\|\mathcal{P}\|m^{\frac{n}{2}}(\frac{n}{2})^{2m+2})$	$O^*(\|\mathcal{P}\|m^{\frac{n}{2}}(\frac{n}{2})^{2m+2})$
$P\|d_j\|C_{\max}, \sum_j w_j U_j$	Shang and T'Kindt [79]	$O^*(\|\mathcal{P}\|(m+1)^{\frac{n}{2}}(\frac{n}{2})^{3m})$	$O^*(\|\mathcal{P}\|(m+1)^{\frac{n}{2}}(\frac{n}{2})^{3m})$

Note: the last two lines refer to multi-objective problems, and $|\mathcal{P}|$ is the number of Pareto optima.

1.3.3 Branching Algorithms

Branching consists in decomposing a problem into subproblems, recursively solving the subproblems and combining their solutions into a solution of the original problem. Each recursion step is represented by a node in a branching tree.

As described by Fomin and Kratsch [23, chap. 4], several branching algorithms use well defined rules, which lead to a precise evaluation of their worst-case complexity bounds. They are called Branch and Reduce algorithms. Their complexity analysis can be refined by the measure and conquer technique [23, chap. 6], enabling to derive better complexity bounds without changing the algorithm. Unfortunately, these algorithms are very rare when applied to scheduling problems as designing, for instance, a reduction rule is not straightforward.

Table 1.20 lists the branching algorithms applied to scheduling with an interesting worst-case complexity bound. Unlike the other tables, it only contains the work of Garraffa et al. [29]. They introduce a node merging mechanism to detect identical sub-problems in a branching tree and thus to avoid solving them multiple times. It is one of the few branching algorithms in scheduling with a proven non-trivial worst-case time complexity bound.

Table 1.20: Time and space worst-case complexity bounds of a scheduling problem solved by a branching algorithm

problem	reference	time	space
$1\|d_j\|\sum_j T_j$	Garraffa et al. [29]	$O^*(2^n)$	$O^*(1)$

Branch-and-Bound are particular branching algorithms, in which each node represents a set of solutions, and where we bound the optimum between a lower bound (LB) and an upper bound (UB). The quality of the lower bound is crucial for the performance of the algorithm.

We have a very poor knowledge about how to analyze the worst-case complexity bounds of Branch and Bound algorithms. Most Branch and Bound algorithms therefore have the same worst-case complexity bounds as brute-force enumeration solutions, *i.e.* in $O^*(n!)$ or $O^*(n^n)$ time. However, Branch-and-Bound algorithms turn out to be state-of-the-art in practice. That's why Branch-and-Bound algorithms are very popular. In their review, Tomazella and Nagano [87] enumerate about 120 Branch and Bound algorithms, only for flowshop problems.

Table 1.21: A review of Branch and Bound algorithms for two sequencing problems

(a) The $1|\tilde{d}_j|\sum w_jC_j$ problem

year	reference
1967	Burns [13]
1980	Bansal [4]
1983	Potts and Van Wassenhove [75]
1985	Posner [73]
1987	Bagchi and Ahmadi [3]
1988	Abdul-Razaq and Potts [1]
2003	Pan [63]
2004	T'kindt et al. [85]
2021	Shang et al. [81]

(b) The $F|prmu|C_{\max}$ problem

year	reference
1965	Ignall and Schrage [37]
1965	Lomnicki [52]
1966	Brown and Lomnicki [11]
1967	McMahon and Burton [54]
1970	Ashour [2]
1978	Lageweg et al. [46]
1980	Potts [74]
1991	Brah and Hunsucker [10]
1996	Carlier and Rebaï [15]
1997	Cheng et al. [16]
2005	Ladhari and Haouari [45]
2016	Ritt [76]
2020	Gmys et al. [30]

Branch-and-Bound is very successful when applied to sequencing problems, *i.e.* scheduling problems reducing to finding a job order. For example, the state-of-the-art exact method concerning the $1|r_j, \tilde{d}_j|$- problem is a Branch-and-Bound algorithm proposed by Carlier [14].

Table 1.21 summarizes the history of Branch-and-Bound algorithms for two emblematic problems: $1|\tilde{d}_j|\sum w_jC_j$ and $F||C_{\max}$. Shang et al. [81] and Gmys et al. [30] proposed the currently fastest known exact algorithms in practice among all algorithms.

1.3.4 Inclusion-Exclusion

We briefly describe the principles of the Inclusion-Exclusion technique. The whole chapter 2 is devoted to an accurate description of this technique and its relatives.

Given a scheduling problem, the Inclusion-Exclusion technique consists in deciding whether there exists or not a solution whose objective value is bounded by a given threshold. Here are its main steps, in principle and omitting the details. (1) Relax the problem by allowing duplicate jobs. (2) Count relaxed solutions when only certain jobs (potentially duplicate) are allowed. (3) Apply the Inclusion-Exclusion formula (a sum over job subsets) to count relaxed solutions which encompass non-relaxed solutions. (4) Thus, decide whether there exists a non-relaxed solution or not. This approach has two advantages: (1) it transforms the solution of a problem with a sequencing part in $O^*(n!)$ into the solution of $O^*(2^n)$ relaxed problems; (2) a relaxed problem can be solved in pseudopolynomial time and space using dynamic programming. Finally, we can derive from the Inclusion-Exclusion technique an algorithm whose worst case complexity bounds are moderately exponential in time and most often pseudo-polynomial in space.

Table 1.22: Time and space worst-case complexity bounds of some scheduling problems solved by Inclusion-Exclusion

problem	reference	time	space
$1\|r_j, \tilde{d}_j\|$-	Karp [42]	$O^*(2^n \|\|\mathcal{I}\|\|)$	$O^*(\|\|\mathcal{I}\|\|)$
$P\|C_{\max} \leqslant B\|$-	Karp [42]	$O^*(2^n B)$	$O^*(B)$
$R\|r_{ji}, \tilde{d}_{ji}\|f_{\max}$	Chapter 3	$O^*(2^n \|\|\mathcal{I}\|\|)$	$O^*(\|\|\mathcal{I}\|\|)$
$R\|r_{ji}, \tilde{d}_{ji}\|\sum_j f_j$	Chapter 3	$O^*(2^n \|\|\mathcal{I}\|\|\gamma^{opt})$	$O^*(\|\|\mathcal{I}\|\|\gamma^{opt})$
$F\|prmu, r_j, \tilde{d}_j\|f_{\max}$	Chapter 4	$O^*(2^n \|\|\mathcal{I}\|\|^m)$	$O^*(\|\|\mathcal{I}\|\|^m)$
$F\|prmu, prec, r_j, \tilde{d}_j\|f_{\max}$	Chapter 4	$O^*(2^n \|\|\mathcal{I}\|\|^m)$	$O^*(2^{n-\mu} \|\|\mathcal{I}\|\|^m)$
$F\|prmu, r_j, \tilde{d}_j\|\sum_j f_j$	Chapter 4	$O^*(2^n \|\|\mathcal{I}\|\|^m \gamma^{opt})$	$O^*(\|\|\mathcal{I}\|\|^m \gamma^{opt})$
$F\|prmu, prec, r_j, \tilde{d}_j\|\sum_j f_j$	Chapter 4	$O^*(2^n \|\|\mathcal{I}\|\|^m \gamma^{opt})$	$O^*(2^{n-\mu} \|\|\mathcal{I}\|\|^m \gamma^{opt})$

Notes: B represents the bin capacity in the bin-packing problem.
γ^{opt} is the optimal objective value, a pseudopolynomial factor.
μ is the number of maximal jobs *i.e.* jobs without successors.

1.4 Conclusions

In this chapter we have shown the interest of exponential algorithmics to solve scheduling problems to optimality.

Numerous scheduling problems are NP-hard. Assuming $\mathcal{P} \neq \mathcal{NP}$, there is no worst-case polynomial time algorithm to solve such problems to optimality. As a consequence, only super-polynomial algorithms can solve such problems to optimality. And most super-polynomial algorithms are actually exponential. Exponential algorithmics aims at designing an exact algorithm, ideally with a worst case time complexity in $O^*(c^n)$ with c as small as possible.

Among the various techniques we have seen, some, like Branch-and-Bound, are often very effective in practice but offer few theoretical guarantees and have high, often factorial, worst-case time complexities. Others, like dynamic programming across subsets or Sort and Search, guarantee exponential time complexities but also have exponential space complexities. Finally, Inclusion-Exclusion guarantees moderate exponential time complexities, together with pseudopolynomial space complexities, which makes it particularly interesting from a theoretical point of view.

We shall apply Inclusion-Exclusion to the very challenging field of scheduling. Chapter 2 is devoted to the study of the Inclusion-Exclusion technique in general. In chapters 3 and 4, we apply this technique to very general classes of parallel machine and flowshop scheduling problems with regular objectives and we derive moderately exponential time and pseudopolynomial space algorithms to solve these problems.

Chapter 2

Inclusion-Exclusion for Scheduling

Topics

- We describe the principles of Inclusion-Exclusion: how to transform a dynamic programming scheme across subsets running in exponential time and space into an Inclusion-Exclusion algorithm running, for most problems, in moderate exponential time and only pseudopolynomial space.
- As an application, we give a detailed description of two representative examples: the Shortest Directed Hamiltonian Path problem and the Interval Sequencing problem.
- We review several techniques to accelerate Inclusion-Exclusion, from a theoretical and practical point of view: Bonferroni inequalities, Möbius inversion, Abstract Tubes, Index space trimming and Zero sweeping.

The Inclusion-Exclusion technique, derived from a rather old combinatorics formula, received a pioneer application to computer science by Kohn et al. [43], Karp [42] and Bax [6]. More recently, Inclusion-Exclusion gained in popularity in Operations Research (see for example Bjorklund and Husfeldt [8], and Koivisto [44]). Nederlof [59, 61] showed the interest of this technique to get polynomial or pseudopolynomial space and moderate exponential time algorithms.

In this chapter, we focus on the application of Inclusion-Exclusion to scheduling. In section 2.1, we describe how to solve an optimization problem to optimality using the Inclusion-Inclusion technique. In section 2.2, we describe two representative examples, the Shortest Directed Hamiltonian Path problem and Interval Sequencing. In section 2.3, we review techniques related to Inclusion-Exclusion and their application to permutation scheduling problems.

2.1 Principles

As we saw in section 1.3.1, many scheduling problems can be solved by dynamic programming across subsets. Unfortunately, we often get algorithms whose memory consumption is exponential. From a theoretical point of view, Inclusion-Exclusion enables to have

comparable time complexities, while maintaining pseudopolynomial space complexity, and without increasing the difficulty of the recurrence equations. This technique is well-suited for coverage problems, particularly for permutation problems.

In section 2.1.1, we show how an optimization problem can be polynomially reduced to a decision problem. Of course, this decision problem itself reduces to a counting problem. In section 2.1.2, we introduce the Inclusion-Exclusion formula in its mathematical version. In section 2.1.3, we introduce the Inclusion-Exclusion formula in its computer science version. We see how to solve a coverage counting problem by relaxing it, and thus how to solve the associated decision problem.

2.1.1 From solution existence to an explicit optimal solution

Let us consider an optimization problem $\mathcal{P}_{opt}$. From the mathematical point of view, this problem is specified by two functions: a function $\mathscr{S}$ which gives all the solutions of the problem, and an objective function γ to be minimized. For any instance $\mathcal{I}$, the aim is to compute an optimal solution $\mathcal{P}_{opt}(\mathcal{I}) = \min_{S \in \mathscr{S}(\mathcal{I})} \gamma(S)$.

The set of solutions $\mathscr{S}(\mathcal{I})$ always has a structure. To simplify the explanations, we will consider that each solution $S \in \mathscr{S}(\mathcal{I})$ is a sequence of ℓ elements, each in $\{1, \ldots, h\}$, *i.e.* $S = (s_1, \ldots, s_\ell)$ with $\forall i, s_i \in \{1, \ldots, h\}$ and that $\mathscr{S}(\mathcal{I}) \subset \{1, \ldots, h\}^\ell$. For example, for a permutation problem one can imagine that $h = \ell = n$ and that $\mathscr{S}(\mathcal{I})$ is the set $\mathfrak{S}_n$ of permutations of n elements, *i.e.* $\mathscr{S}(\mathcal{I}) = \mathfrak{S}_n \subset \{1, \ldots, n\}^n$. We also assume that h and ℓ are polynomial with respect to the instance size.

In theory, to solve $\mathcal{P}_{opt}$, we can use the brute-force algorithm consisting in listing all the solutions and choosing the best one. For this, we can simply list all the first possible elements, then recursively all possible second elements, and so on. We obtain a search tree in which each node is the set of sequences that start with a certain prefix π, which we denote by $\{\pi*\}$. In this tree, each leaf node corresponds to a complete solution S. Figure 2.1 shows the branching tree we get.

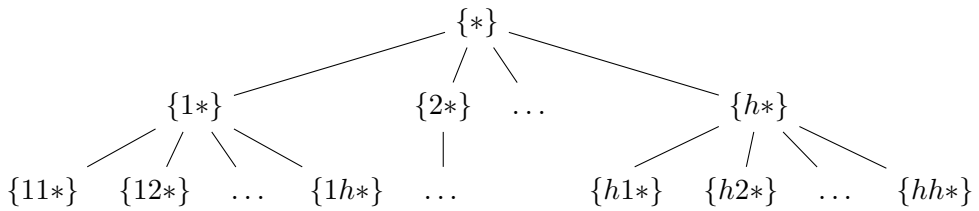


Figure 2.1: Forward branching tree of solutions of a sequencing problem

The brute-force algorithm enumerates all nodes in the tree, so it has a time complexity in $\Theta^*(\operatorname{card} \mathscr{S}(\mathcal{I}))$, which is often of the order of magnitude of $n!$ or n^n. To avoid this issue, we will use a well suited decision problem as an oracle to know whether to cut a node or whether to explore it.

We now define the decision problem $\mathcal{P}_{dec}$ associated with $\mathcal{P}_{opt}$ in a more precise version than usual: given an instance $\mathcal{I}$, a prefix π and a threshold value of the objective ε, $\mathcal{P}_{dec}(\mathcal{I}, \pi, \varepsilon)$ is true when there exists a solution S with prefix π and objective value of at most ε. In

other words:

$$\mathcal{P}_{dec}(\mathcal{I}, \pi, \varepsilon) \iff \exists\, S \in \mathscr{S}(\mathcal{I}) \cap \{\pi*\} \mid \gamma(S) \leqslant \varepsilon \tag{2.1}$$

Note that this is an existence problem, *i.e.* an implicit problem: we are not trying to build an explicit solution S, but we are only trying to know whether there exists one or not. Note also that the decision problem usually associated with an optimization problem does not take any prefix into account, which is equivalent to consider this prefix as empty. The usual problem is denoted by $\mathcal{P}_{dec}(\mathcal{I}, \varepsilon)$, it is defined by

$$\mathcal{P}_{dec}(\mathcal{I}, \varepsilon) \iff \exists\, S \in \mathscr{S}(\mathcal{I}) \mid \gamma(S) \leqslant \varepsilon \tag{2.2}$$

and we obviously have $\mathcal{P}_{dec}(\mathcal{I}, \varepsilon) \iff \mathcal{P}_{dec}(\mathcal{I}, (), \varepsilon)$, where $()$ denotes the empty prefix.

As a decision problem, $\mathcal{P}_{dec}$ can be straightforwardly reduced to a counting problem $\mathcal{P}_{\#}$, corresponding to a counting function N. We introduce the following definition:

Definition 2.1. We call an implementation of the decision problem $\mathcal{P}_{dec}$ any function N computing an integer such that

$$\forall \mathcal{I}, \pi, \varepsilon,\ \mathcal{P}_{dec}(\mathcal{I}, \pi, \varepsilon) \iff N(\mathcal{I}, \pi, \varepsilon) > 0 \tag{2.3}$$

As an extension, the parameters of N do not need to match exactly the parameters of $\mathcal{P}_{dec}$ as in this equation. Of course, it is not necessary in general to reduce a decision problem to a counting problem, but that is what we will do every time we use the Inclusion-Exclusion technique.

Because $\mathcal{P}_{dec}$ is an existence problem, the most straightforward implementation of $\mathcal{P}_{dec}$ consists in determining the very precise number N of solutions S selected by $\mathcal{P}_{dec}$, that is:

$$N(\mathcal{I}, \pi, \varepsilon) = \mathrm{card}\{S \in \mathscr{S}(\mathcal{I}) \cap \{\pi*\} \mid \gamma(S) \leqslant \varepsilon\} \tag{2.4}$$

but this is not the only possible implementation. For example, in chapter 3, our solution of parallel machine scheduling problems is based on an alternate implementation.

Reducing $\mathcal{P}_{opt}$ to $\mathcal{P}_{dec}$

Let us suppose that we know how to solve the decision problem $\mathcal{P}_{dec}$ associated with $\mathcal{P}_{opt}$. We will show that we can solve $\mathcal{P}_{opt}$, *i.e.* compute an explicit optimal solution, using only a polynomial number of calls to $\mathcal{P}_{dec}$.

First, we can determine the optimum objective value γ^{opt}. It is:

$$\gamma^{opt} = \min\{\varepsilon \in \mathbb{N} \mid \mathcal{P}_{dec}(\mathcal{I}, (), \varepsilon)\} \tag{2.5}$$

Algorithm 2.2 p. 48 details the concrete calculation of $\gamma^{opt} = \mathit{OptimumObjective}(\mathcal{I})$. Stage 1 is used to enclose γ^{opt} between a lower bound $\varepsilon_{\min}$ and an upper bound $\varepsilon_{\max}$, but if we already know bounds we can use them instead. Stage 2 is a simple binary search. Note that this algorithm performs a polynomial number of steps. Indeed, since γ^{opt} is polynomial with respect to $||\mathcal{I}||$ there are $O(\log \gamma^{opt}) = O(\log ||\mathcal{I}||) = O^*(1)$ steps.

Algorithm 2.2: Computation of the optimum objective value γ^{opt}

```
Function OptimumObjective(I):
    // Stage 1: bound γ^opt
    ε_max ← 1
    while P_dec(I, (), ε_max) = 0 do
        ε_max ← 2 ε_max
    ε_min ← 0
    // Stage 2: binary search
    while ε_min < ε_max do
        ε ← ⌈½(ε_min + ε_max)⌉
        if P_dec(I, (), ε) > 0 then
            [ε_min, ε_max] ← [ε_min, ε]
        else
            [ε_min, ε_max] ← [ε + 1, ε_max]
    return ε_min    // γ^opt = ε_min = ε_max
```

Algorithm 2.3 is used to compute an optimal solution. We denote sequences by $(s_1, s_2, \ldots)$ and the sequence concatenation operator by $(\cdot)$. After calculating γ^{opt}, we use $\mathcal{P}_{dec}(\mathcal{I}, \pi, \gamma^{opt})$ as an oracle to predict whether a search tree path passing through the node $\{\pi*\}$ leads to an optimal solution or not, and we can immediately explore or cut this node. As a loop invariant, it is guaranteed that the search for an element s to add to π is successful. Finally, we walk in the search tree, without ever backtracking, through a path of length ℓ from the root to a leaf node corresponding to an optimal solution.

Algorithm 2.3: Computation of an optimal solution

```
Function OptimalSolution(I):
    γ^opt ← OptimumObjective(I)
    π ← ()
    repeat ℓ times
        find s ∈ {1, ..., h} such that P_dec(I, π · (s), γ^opt)
        π ← π · (s)
    return π
```

Remark 2.2. Due to the structure of Algorithm 2.3, the oracle $\mathcal{P}_{dec}$ is always called with an admissible prefix π, *i.e.* there always exists a suffix σ such that $\pi \cdot \sigma$ is a solution. So, an implementation of $\mathcal{P}_{dec}$ never needs to test the admissibility constraints of the problem on π, they are automatically answered.

Clearly, the number of calls to $\mathcal{P}_{dec}$ in Algorithm 2.3 is in $O(\log \gamma^{opt}) = O^*(1)$ for the computation of γ^{opt} and in $O(h\ell) = O^*(1)$ for the computation of the optimal solution: it is therefore polynomial, which justifies the following proposition.

Proposition 2.3. An optimal solution of $\mathcal{P}_{opt}$ can be computed using a polynomial number of calls to $\mathcal{P}_{dec}$, *i.e.* $\mathcal{P}_{opt} \propto \mathcal{P}_{dec}$.

We now define the concept of self-reducibility:

Definition 2.4. A problem is said to be self-reducible when the solution of an instance can be reduced to the solution of a simpler instance of the same problem. Put otherwise, a problem is self-reducible when it can be expressed using simple recursive equations.

Remark 2.5. The key point is to determine the effect of appending an element v to a prefix π, *i.e.* to find a simple recurrence relation of the form $\mathcal{P}_{dec}(\mathcal{I}, \pi \cdot (v), \varepsilon) \Longleftrightarrow \mathcal{P}_{dec}(\mathcal{I}', \pi, \varepsilon')$ for some $\mathcal{I}'$ and ε'. Such a relation often enables at the same time not only to define a dynamic programming scheme across subsets, but also to define a dynamic programming scheme to count admissible relaxed lists, and finally to implement the decision problem and solve the optimization problem by Inclusion-Exclusion.

Notice that Fomin and Kratsch [23, chap. 4] define self-reduction as any method enabling to derive an optimal solution from a polynomial number of calls to a counting function as N, usually implemented via Inclusion-Exclusion. Proposition 2.3 p. 48 combined with Definition 2.1 p. 47 shows that any self-reducible problem in the sense of Definition 2.4 leads to a self-reduction method as defined by Fomin and Kratsch.

For the sake of explanations, we have taken the particular case where the solutions are sequences, but in the general case the explanations remain valid whenever we can define notions of concatenation, prefix and suffix of solutions. For example, in chapter 3, we define a special concatenation in the case of scheduling with parallel machines.

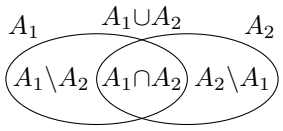
We also focused on solution prefixes, which correspond to forward branching. This is not the only possibility as we can also branch backwards, but we have chosen this presentation because it is more intuitive in scheduling, where it is common to proceed in chronological order.

2.1.2 The Inclusion-Exclusion formula

The Inclusion-Exclusion formula is attributed to the French mathematician Abraham de Moivre (1667–1754). It enables to express the cardinal of a union of sets as an alternating sum of the cardinals of their partial intersections. All the sets that we will handle are implicitly finite.

For two sets A_1, A_2 we have this well-known formula, deduced from the partition into regions of the Venn diagram representing the union of the two sets (Figure 2.4):

$$\text{card}(A_1 \cup A_2) = \text{card}(A_1) + \text{card}(A_2) - \text{card}(A_1 \cap A_2) \tag{2.6}$$



$$\begin{cases} \text{card} \quad (A_1) \quad = \text{card}(A_1 \backslash A_2) + \text{card}(A_1 \cap A_2) \\ \text{card} \quad (A_2) \quad = \qquad\qquad\quad \text{card}(A_1 \cap A_2) + \text{card}(A_2 \backslash A_1) \\ \text{card}\ (A_1 \cup A_2) = \text{card}(A_1 \backslash A_2) + \text{card}(A_1 \cap A_2) + \text{card}(A_2 \backslash A_1) \end{cases}$$

Figure 2.4: Venn diagram to relate cardinals of union and intersection of two sets

For 3 sets A_1, A_2 and A_3 we have the following formula:

$$\begin{aligned} \text{card}(A_1 \cup A_2 \cup A_3) \ = \ & + \text{card}(A_1) + \text{card}(A_2) + \text{card}(A_3) \\ & - \text{card}(A_1 \cap A_2) - \text{card}(A_1 \cap A_3) - \text{card}(A_2 \cap A_3) \\ & + \text{card}(A_1 \cap A_2 \cap A_3) \end{aligned} \tag{2.7}$$

which can be intuitively interpreted as:

$$\begin{aligned}\text{cardinal of union of 3 sets} \;=\; &+ \text{ sum of cardinals of intersections of 1 set}\\ &- \text{ sum of cardinals of intersections of 2 sets}\\ &+ \text{ sum of cardinals of intersections of 3 sets}\end{aligned}$$

This formula can be generalized. To state the general case, we use sets A_i containing elements, indices i and sets of indices I. We denote $\mathrm{card}(A_i)$ when we count elements but $|I|$ when we count indices. We make this distinction for two reasons: first, from a mathematical point of view, the cardinal of a set A_i is proportional to a probability measure $\mathbb{P}$, and the formulas remain valid when we change this measure. Second, from a computational point of view, index sets are limited in size. This size is much less than the capacity of a machine word, so indices can be typed as machine native integers, whereas the cardinals of sets A_i that we manipulate can take very large values and should be typed as big integers. Here is the Inclusion-Exclusion formula in the general case.

Proposition 2.6: Inclusion-Exclusion formula.
The cardinal of the union of sets is the alternating sum of the cardinals of their partial intersections, excluding the empty intersection. For sets $A_1, \ldots, A_n$, we have:

$$\mathrm{card}\Big(\bigcup_{i=1}^{n} A_i\Big) = \sum_{\substack{I\subset\{1,\ldots,n\}\\ I\neq\varnothing}} (-1)^{|I|-1}\,\mathrm{card}\Big(\bigcap_{i\in I} A_i\Big) \tag{2.8}$$

Considering the family $(A_i)_{i\in J}$ with $|J| = n$, the formula becomes:

$$\mathrm{card}\Big(\bigcup_{i\in J} A_i\Big) = \sum_{\substack{I\subset J\\ I\neq\varnothing}} (-1)^{|I|-1}\,\mathrm{card}\Big(\bigcap_{i\in I} A_i\Big) \tag{2.9}$$

Although this result is well known (see e.g. Mazur [53]), for the sake of completeness we provide a proof of its correctness.

Proof. We prove the result by induction on n. It's trivially true for $n = 0$ ($0 = 0$) and for $n = 1$ ($\mathrm{card}(A_1) = \mathrm{card}(A_1)$). Now, we suppose it is true for n and we prove it for $n + 1$.

First, notice that for all sets A_i and B, the value of the union $\big(\bigcup_i A_i \cap B\big)$ does not depend on parenthesis grouping: $(\bigcup_i A_i) \cap B = \bigcup_i (A_i \cap B) = \bigcup_i A_i \cap B$. Similarly, for a (non-empty) intersection: $(\bigcap_i A_i) \cap B = \bigcap_i (A_i \cap B) = \bigcap_i A_i \cap B$. Thus, we apply the two-set Formula (2.6) p. 49 to both sets $\bigcup_{i=1}^{n} A_i$ and A_{n+1}. Then, we apply twice the recurrence hypothesis: to $\bigcup_{i=1}^{n} A_i$ on the left, and to $\bigcup_{i=1}^{n} (A_i \cap A_{n+1})$ on the right. We derive:

$$\begin{aligned}
&\mathrm{card}\Big(\bigcup_{i=1}^{n} A_i \cup A_{n+1}\Big)\\
=\;&\mathrm{card}\Big(\bigcup_{i=1}^{n} A_i\Big) &&+\; \mathrm{card}\Big(A_{n+1}\Big) &&-\; \mathrm{card}\Big(\bigcup_{i=1}^{n} A_i \cap A_{n+1}\Big)\\
=\;&\sum_{\substack{I\subset\{1,\ldots,n\}\\ I\neq\varnothing}} (-1)^{|I|-1}\,\mathrm{card}\Big(\bigcap_{i\in I} A_i\Big) &&+\; \mathrm{card}\Big(A_{n+1}\Big) &&-\; \sum_{\substack{I'\subset\{1,\ldots,n\}\\ I'\neq\varnothing}} (-1)^{|I'|-1}\,\mathrm{card}\Big(\bigcap_{i\in I'} A_i \cap A_{n+1}\Big)
\end{aligned}$$

On the left, we rearrange the set of sum indices without changing it. In the middle, we write the single term as a sum. On the right, we make the variable change $I = I' \cup \{n+1\} \Leftrightarrow I' = I \setminus \{n+1\}$, and we have $-(-1)^{|I'|-1} = +(-1)^{|I|-1}$. Sets of sum indices combine

nicely and we get:

$$\begin{aligned}
&\operatorname{card}\Big(\bigcup_{i=1}^{n} A_i \cup A_{n+1}\Big) \\
= &\sum_{\substack{I\subset\{1,\dots,n+1\}\\ n+1\notin I\\ I\neq\varnothing}} (-1)^{|I|-1} \operatorname{card}\Big(\bigcap_{i\in I} A_i\Big) \;+\; \sum_{\substack{I\subset\{1,\dots,n+1\}\\ n+1\in I\\ I=\{n+1\}}} (-1)^{|I|-1} \operatorname{card}\Big(\bigcap_{i\in I} A_i\Big) \;+\; \sum_{\substack{I\subset\{1,\dots,n+1\}\\ n+1\in I\\ I\neq\{n+1\}}} (-1)^{|I|-1} \operatorname{card}\Big(\bigcap_{i\in I} A_i\Big) \\
= &\sum_{\substack{I\subset\{1,\dots,n+1\}\\ I\neq\varnothing}} (-1)^{|I|-1} \operatorname{card}\Big(\bigcap_{i\in I} A_i\Big)
\end{aligned}$$

□

It is often preferable to express unions as intersections of complements. So, let E be a (finite) universe containing all A_i and we define, as usual, the complement of A_i by $\bar{A}_i = E \setminus A_i$. An empty intersection is the universe: $\bigcap_{i\in\varnothing} A_i = E$. We derive the following proposition.

Proposition 2.7: Inclusion-Exclusion, alternate form.
The cardinal of an intersection of complements is expressed as an alternating sum of the cardinals of their partial intersections, including the empty intersection. For a universe E and $(A_i)_{i\in J} \subset E$, we have:

$$\operatorname{card}\Big(\bigcap_{i\in J} \bar{A}_i\Big) = \sum_{I\subset J} (-1)^{|I|} \operatorname{card}\Big(\bigcap_{i\in I} A_i\Big) \tag{2.10}$$

Proof. We have:

$$\begin{aligned}
\operatorname{card}\Big(\bigcap_{i\in J} \bar{A}_i\Big) &= \operatorname{card}\Big(E\Big) - \operatorname{card}\Big(\bigcup_{i\in J} A_i\Big) \\
&= \operatorname{card}\Big(\bigcap_{i\in\varnothing} A_i\Big) - \sum_{\substack{I\subset J\\ I\neq\varnothing}} (-1)^{|I|-1} \operatorname{card}\Big(\bigcap_{i\in I} A_i\Big) \\
&= \sum_{\substack{I\subset J\\ I=\varnothing}} (-1)^{|I|} \operatorname{card}\Big(\bigcap_{i\in I} A_i\Big) + \sum_{\substack{I\subset J\\ I\neq\varnothing}} (-1)^{|I|} \operatorname{card}\Big(\bigcap_{i\in I} A_i\Big) \\
&= \sum_{I\subset J} (-1)^{|I|} \operatorname{card}\Big(\bigcap_{i\in I} A_i\Big)
\end{aligned}$$

□

2.1.3 The relaxation principle

An important use case of Inclusion-Exclusion consists in applying it to data lists and solving coverage problems, and among them, permutation problems. Before developing further, we need some technical paragraphs to formally define, from a mathematical point of view, the lists to which Inclusion-Exclusion can be applied.

A list is a collection of elements spotted by a position (we reserve the term "index" for labeling sets involved in Inclusion-Exclusion). From a mathematical point of view, a list L is a family $(L_p \in V)_{p\in S}$, *i.e.* a function which to each position p associates a value L_p

in some set V. The set S of positions is the shape of the list (e.g. $S = \{1, \dots, n\}$ for a sequence of length n and $S = \{1, \dots, n\} \times \{1, \dots, m\}$ for a $n \times m$ matrix). We denote (as usual) by V^S the set of lists of shape S on the set V.

We often need to manipulate sets of lists of variable shape on a set V. It is just a (disjoint) union of sets of fixed-shape lists on V. In other words, we introduce a set $\mathbb{S}$ of possible shapes, and we consider the set $\bigcup_{S \in \mathbb{S}} V^S$. In particular, we define the set of linear lists (sequences) of length exactly n on V by $V^n = V^{\{1,\dots,n\}}$ and the set of lists of length at most n on V by $V^{(n)} = \bigcup_{\ell=0}^{n} V^\ell$, which amounts to setting $\mathbb{S} = \big\{\{1, \dots, \ell\}, 0 \leqslant \ell \leqslant n\big\}$ and $V^{(n)} = \bigcup_{S \in \mathbb{S}} V^S$.

For Inclusion-Exclusion, the precise shape of the lists does not matter. From now on, we write V^* to mean that we have chosen once and for all a set $\mathbb{S}$ of shapes and that we have defined $V^* = \bigcup_{S \in \mathbb{S}} V^S$.

As an extension of set notation, we write $v \in L$ to mean that the value v appears in the list L, in other words: $v \in (L_p)_{p \in S} \iff \exists p \in S \mid L_p = v \iff v \in \{L_p, p \in S\}$. We also define covering lists as lists where all the values appear (these lists are also the surjective functions from S to V). The set of covering lists is denoted by $\mathscr{C}_V$, or just $\mathscr{C}$ whenever there is no ambiguity. It is defined by:

$$\mathscr{C} = \{L \in V^* \mid \forall\, v \in V, v \in L\} \tag{2.11}$$

Counting covering lists

Consider a counting problem $\mathcal{P}_\#$ (similar to the one defined in section 2.1.1), and suppose that it is also a coverage problem: there is some set V to cover, and $\mathcal{P}_\#$ counts covering lists on V, which in addition satisfy a given admissibility criterion $\mathscr{A}$ specific to the $\mathcal{P}_\#$ problem.

So, solving $\mathcal{P}_\#$ leads to compute $\mathrm{card}(\mathscr{S})$ where $\mathscr{S}$ is the set of covering admissible lists, which we will also call strict lists: $\mathscr{S} = \{L \in \mathscr{C} \mid \mathscr{A}(L)\}$. Let us now relax the coverage constraint and define the set $\mathscr{R}$ of relaxed admissible lists by: $\mathscr{R} = \{L \in V^* \mid \mathscr{A}(L)\}$. Obviously, we have: $\mathscr{S} = \mathscr{R} \cap \mathscr{C}$, *i.e.* the strict admissible lists are the covering relaxed admissible lists.

We will now apply the Inclusion-Exclusion formula. We take as universe the set of relaxed admissible lists: $E = \mathscr{R}$, and we consider the sets $(A_v)_{v \in V}$ defined by: $\forall\, v, A_v = \{L \in \mathscr{R} \mid v \notin V\}$. Note that the indices of the Inclusion-Exclusion formula are the values appearing in the lists, *i.e.* the elements of V. After variable renaming, the alternate formula (2.10) p. 51 transforms into:

$$\mathrm{card}\Big(\bigcap_{v \in V} \bar{A}_v\Big) = \sum_{W \subset V} (-1)^{|W|}\, \mathrm{card}\Big(\bigcap_{v \in W} A_v\Big) \tag{2.12}$$

On the left, we have $\bar{A}_v = \{L \in \mathscr{R} \mid v \in V\}$, so

$$\begin{aligned} \bigcap_{v \in V} \bar{A}_v &= \{L \in \mathscr{R} \mid \forall\, v \in V, v \in L\} \\ &= \{L \in \mathscr{R} \mid L \in \mathscr{C}\} \\ &= \mathscr{R} \cap \mathscr{C} \end{aligned} \tag{2.13}$$

On the right, we have:

$$\begin{aligned}\bigcap_{v\in W} A_v &= \{L \in \mathscr{R} \mid \forall\, v \in W, v \notin L\}\\ &= \{L \in \mathscr{R} \mid \forall\, v \in L, v \notin W\}\\ &= \{L \in \mathscr{R} \mid \forall\, v \in L, v \in V \setminus W\}\\ &= \mathscr{R} \cap (V \setminus W)^*\end{aligned} \tag{2.14}$$

By replacing these intersections by their values in Formula (2.12) p. 52, we get:

$$\text{card}\ \underbrace{(\mathscr{R} \cap \mathscr{C})}_{\substack{\text{covering}\\ \text{admissible lists}}} = \sum_{W \subset V} (-1)^{|W|}\ \text{card}\ \underbrace{(\mathscr{R} \cap (V \setminus W)^*)}_{\substack{\text{relaxed admissible lists}\\ \text{with no value in } W}} \tag{2.15}$$

In the sequel, we find it easier to consider relaxed admissible lists using only (optional) values of a subset X instead of relaxed admissible lists with no value in a subset W (*i.e.* no withdrawn value), so we apply the variable change $X = V \setminus W \Leftrightarrow W = V \setminus X$, and we have $|W| = |V| - |X|$. We derive:

$$\text{card}\ \underbrace{(\mathscr{R} \cap \mathscr{C})}_{\substack{\text{covering}\\ \text{admissible lists}}} = \sum_{X \subset V} (-1)^{|V|-|X|}\ \text{card}\ \underbrace{(\mathscr{R} \cap X^*)}_{\substack{\text{relaxed admissible lists}\\ \text{using only values in } X}} \tag{2.16}$$

It is convenient to define N as $\text{card}(\mathscr{R} \cap \mathscr{C})$ and, for each subset X of V, N_X as $\text{card}(\mathscr{R} \cap X^*)$. Finally, we derive the following proposition.

Proposition 2.8: Relaxation Principle. Consider a problem $\mathcal{P}_\#$ that relates to counting admissible lists which cover a set V. Relax the coverage constraint, and calculate, for all $X \subset V$, the number N_X of admissible lists, covering or not, using only elements of X. Then, the number N of covering admissible lists is given by:

$$\underset{\substack{\uparrow\\ \#\text{ covering}\\ (i.e.\text{ strict})\\ \text{admissible lists}}}{N} = \sum_{X \subset V} (-1)^{|V|-|X|} \underbrace{N_X}_{\substack{\uparrow\\ \#\text{ admissible lists}\\ (i.e.\text{ relaxed})\\ \text{using only values in } X}} \tag{2.17}$$

We have detailed the rationale using the sets W of withdrawn values because it is commonly used in the literature (see e.g. Fomin and Kratsch [23] or Nederlof [59]). Here is another way of thinking: in V^* the covering lists are the lists L which do not belong to any X^* with $X \subsetneq V$. Indeed, the lists of X^* are the lists that do not hit the set $(V \setminus X)$. So, the covering lists are the elements of the universe which are not in the union of X^*, $X{=}V$ excluded, and we have:

$$\mathscr{R} \cap \mathscr{C} \quad = \quad \mathscr{R} \quad \setminus \bigcup_{X \subsetneq V} \mathscr{R} \cap X^* \tag{2.18}$$

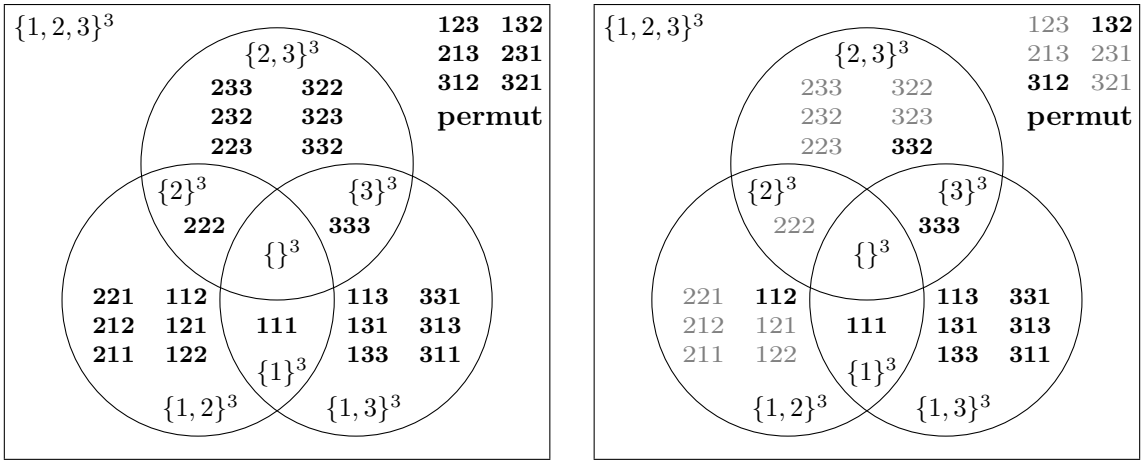
This formula becomes very intuitive when illustrated by a Venn diagram, as in Figure 2.5 p. 54. It enables to directly derive the formula (2.16) by applying the Inclusion-Exclusion formula (2.8) p. 50.

Counting permutations

Among covering lists, permutations are a very useful particular case. When viewed as a list, a permutation on a set V is a list of elements, *i.e.* a family $L = (v_p \in V)_{p \in V}$, which is covering ($\forall\, v, \exists\, p \mid v = v_p$) and without duplicate ($p \neq p' \Rightarrow v_p \neq v_{p'}$). Note that V constitutes both the set of positions and values: $L \in V^V$. We denote by $\mathfrak{S}_V$ the set of permutations on V, and by $\mathfrak{S}_n$ the set of permutations on $\{1, \ldots, n\}$.

We just defined a permutation as a covering list and without duplicates, but this definition is redundant: for a list of V^V it is equivalent to be covering and without duplicates (just like for a function from V to V, it is equivalent to be surjective and injective). As a consequence, there is no need to consider duplicates, and a permutation is simply a covering list of V^V. Thus, a relaxed admissible permutation is an ordinary list, with no constraint other than admissibility, which gives the relaxed admissible permutation sets very simple structures.

Figure 2.5 illustrates counting permutations of 3 elements which are the covering lists of V^3 where $V = \{1, 2, 3\}$. The figure does not reproduce the Venn diagram of all $X \subset V$ but the Venn diagram of all $X^3, X \subset V$, which has the same structure because for $X_1, X_2 \subset V$, $X_1^3 \subset X_2^3 \iff X_1 \subset X_2$. The entire universe V^3 is represented by the outer square.



(a) all permutations $\pi = (\pi_1, \pi_2, \pi_3)$ (b) permutations π such that π_1 and π_2 are odd

Figure 2.5: Counting of permutations via Inclusion-Exclusion

In Figure 2.5a we count all the permutations, so we take as set of relaxed (admissible) lists $\mathscr{R} = V^3$, and the number N of (admissible) permutations is given by:

$$\begin{array}{rcccccccccc} N &=& +N_{\{1,2,3\}} & -N_{\{1,2\}} & -N_{\{1,3\}} & -N_{\{2,3\}} & +N_{\{1\}} & +N_{\{2\}} & +N_{\{3\}} & -N_{\{\}} & \\ &=& +27 & -8 & -8 & -8 & +1 & +1 & +1 & -0 & = \ 6 \end{array}$$

In figure 2.5b we count the permutations π such that π_1 and π_2 are odd, so we take as set of relaxed admissible lists $\mathscr{R} = \{(v_1, v_2, v_3) \in V^3 \mid v_1, v_2 \text{ odd}\} = \{1, 3\} \times \{1, 3\} \times \{1, 2, 3\}$, and the number N of admissible permutations is given by:

$$\begin{array}{rcccccccccc} N &=& +N_{\{1,2,3\}} & -N_{\{1,2\}} & -N_{\{1,3\}} & -N_{\{2,3\}} & +N_{\{1\}} & +N_{\{2\}} & +N_{\{3\}} & -N_{\{\}} & \\ &=& +12 & -2 & -8 & -2 & +1 & +0 & +1 & -0 & = \ 2 \end{array}$$

Of course, the special case $n = 3$ is only useful for demonstration purposes. In the general case of permutations on n values, the formula transforms a counting of permutations (in $n!$ steps by brutal enumeration) into 2^n countings on X^* sets. So, this formula is useful when you can efficiently count the particular elements of each set X^*. For example, if every counting is pseudopolynomial, we obtain a moderately exponential complexity instead of a factorial one. The challenge is to achieve this.

2.2 Application examples

The most classic example of the application of Inclusion-Exclusion is the Shortest Directed Hamiltonian Path problem. Note that this problem is very close to the Asymmetric Traveling Salesman problem, himself a direct generalization of the $F|prmu, nowait|C_{\max}$ problem, as we have seen in Example 1.5 p. 30. We also study the Interval Sequencing $1|r_j, \tilde{d}_j|$- problem, seen in Example 1.1 p. 28, which can be considered as the prototype of all scheduling problems for Inclusion-Exclusion.

The first example deals with list of vertices whereas the second example deals with list of jobs, but notice how similar the solution methods for these two examples are. Both examples are self-reducible, which implies that we can derive a dynamic programming scheme across subsets and an Inclusion-Exclusion based algorithm.

Inclusion-Exclusion does not dictate how to count solutions of relaxed problems, and any counting technique can be used a priori. But, as we will see in chapters 3 p. 73 and 4 p. 91, in the case of scheduling problems, the main technique consists in counting relaxed schedules by dynamic programming, in pseudopolynomial time and space for each relaxed problem, even though it may be combined with other techniques, as e.g. generating series convolution.

To the best of our knowledge, the only problem whose counting of relaxed solutions admits an alternative to dynamic programming consists in finding a Hamiltonian path in a directed graph (with unweighted arcs). To solve this problem, Bax [5, 6] defines (the usual way) the adjacency matrix M of a graph as follows: for two vertices u and v, $M_{uv} = 1$ if (u, v) is an arc, and 0 otherwise. Then, the number of paths of length k from u to v is the coefficient $(M^k)_{uv}$ of the k-th power of M. As this method is dedicated to unweighted graphs, it is not directly suitable for optimization problems, which are related to weighted graphs. So, we will not develop it.

2.2.1 Shortest Directed Hamiltonian Path

The Shortest Directed Hamiltonian Path problem can be stated as follows: let $G = (V, A)$ be a directed graph with vertices in V and arcs in A. Consider a distance $d(v, v') \in \mathbb{N}^*, \forall (v, v') \in A$. Set $n = |V|$, and $d(v, v') = +\infty$ if $(v, v') \notin A$. Let $s \neq t \in V$ be a starting vertex and a terminating vertex. The problem consists in finding a Hamiltonian path from s to t, *i.e.* a n-vertex list $(v_1{=}s, v_2, \ldots, v_n{=}t)$ with $\{v_1, \ldots, v_n\} = V$, which minimizes the distance $\sum_i d(v_i, v_{i+1})$. We represent an instance $\mathcal{I}$ of this problem as a tuple $\mathcal{I} = (V, A, d, s, t)$.

Dynamic programming across subsets

We describe the dynamic programming scheme across the subsets of V, which is the traditional algorithm by Held and Karp [35]. Let $Opt[u, X]$ be the minimal distance from vertex u to the terminating vertex t when X is the set of intermediate vertices in any order. We are interested in paths from s to t, so the minimal distance we are looking for is $Opt[s, V \backslash \{s, t\}]$. We have:

$$Opt[u, \varnothing] = d(u, t) \tag{2.19}$$

$$Opt[u, X] = \min_{v \in X}(d(u, v) + Opt[v, X \setminus \{v\}]) \qquad \text{for } X \neq \varnothing \tag{2.20}$$

To obtain a solution of the problem, we compute the backtrace of the dynamic programming scheme. It is a state graph whose arcs are labeled by vertices, of the form $[u, X] \xleftarrow{v^*} [u', X']$. Each time we compute a minimum in Equation (2.20), we keep track of a vertex v^* for which the minimum is reached, *i.e.* $v^* = \arg\min Opt$. Finally, we obtain a (reversed) critical path of the form $[s, V \backslash \{s, t\}] \xleftarrow{v_2} \cdots \xleftarrow{v_{n-1}} [v_{n-1}, \varnothing]$, and we deduce a solution $(s, v_2, \ldots, v_{n-1}, t)$.

Inclusion-Exclusion

The Inclusion-Exclusion method to solve the Shortest Directed Hamiltonian Path problem (or its direct relative, the Asymmetric Traveling Salesman problem) has been described independently by Kohn et al. [43], Karp [42], and Bax [5, 6].

As shown in section 2.1.1, in order to solve the Shortest Directed Hamiltonian Path problem $\mathcal{P}_{opt}$, it is sufficient to solve the decision problem $\mathcal{P}_{dec}(\mathcal{I}, \pi, \varepsilon)$ defined by Equation (2.1) p. 47, where $\mathcal{I}$ is an instance, π is a path prefix containing vertices and ε is a threshold distance.

We now implement the $\mathcal{P}_{dec}$ decision problem, as in Definition 2.1 p. 47. Given a vertex subset $V' \subset V$, a vertex u, and a threshold distance ε, we define $N(u, V', \varepsilon)$ as the number of Hamiltonian paths from u to t, whose set of intermediate vertices is V', whose total distance is lower or equal to ε. Imposing a prefix amounts to subtracting the distance of π from ε and to impose that the remaining vertices but u and t are out of the prefix. We derive:

$$\mathcal{P}_{dec}\Big(\mathcal{I},\ (\pi_1, \ldots, \pi_h),\ \varepsilon\Big) \iff N\Big(\pi_h,\ V \setminus \pi,\ \varepsilon - \sum_{i=1}^{h-1} d(\pi_i, \pi_{i+1})\Big) > 0 \tag{2.21}$$

The paths counted by $N(u, V', \varepsilon)$ are permutations. We relax the permutation constraint, equivalent to a coverage constraint. The set of relaxed paths $\mathscr{R}$ is composed of all paths from u to t with missing or duplicate vertices (*i.e.* loops). Notice that taking $d(v, v') = +\infty$ if $(v, v') \notin A$ ensures that all (v_i, v_{i+1}) are in A. We take $n' = |V'|$ and we derive:

$$\mathscr{R} = \Big\{(v_1, \ldots, v_{n'}) \in V^{n'} \;\Big|\; \sum_{i=0}^{n'} d(v_i, v_{i+1}) \leqslant \varepsilon \text{ where } v_0 = u \text{ and } v_{n'+1} = t\Big\} \tag{2.22}$$

Let $N_X[u, \ell, \varepsilon]$ be the number of (relaxed) paths from u to t, with ℓ intermediate vertices, all in X, and with total distance at most ε. We derive from Formula (2.17) p. 53:

$$N(u, V', \varepsilon) = \sum_{X \subset V'} (-1)^{|V'|-|X|} N_X[u, n', \varepsilon] \tag{2.23}$$

We now just have to compute $N_X[u, \ell, \varepsilon]$ by dynamic programming. For $\ell = 0$ there is only one possible path, from u to t, provided it is short enough. For $\ell > 0$, we try all possible vertices v in first intermediate position. The distance $d(u, v)$ is removed from ε and we recursively count suffixes. Once again, notice that taking $d(v, v') = +\infty$ if $(v, v') \notin A$ ensures that our relaxed paths are actually formed of arcs of the graph. We derive:

$$N_X[u, 0, \varepsilon] = \text{ 1 if } d(u, t) \leqslant \varepsilon, \text{ 0 otherwise} \tag{2.24}$$

$$N_X[u, \ell, \varepsilon] = \sum_{\substack{v \in X \\ d(u,v) \leqslant \varepsilon}} N_X[v, \ell-1, \varepsilon-d(u, v)] \qquad \text{for } \ell > 0 \tag{2.25}$$

As we can see, Formula (2.25) is derived from the same idea as the dynamic programming formula (2.20) p. 56. However, the computation of N is in $O^*(||\mathcal{I}||)$ time and space, where $||\mathcal{I}||$ is the sum of the distances. Therefore, Inclusion-Exclusion solves the problem of the Shortest Directed Hamiltonian Path problem in $O^*(2^n||\mathcal{I}||)$ time and $O^*(||\mathcal{I}||)$ *i.e.* pseudopolynomial space, while dynamic programming across subsets solves it in $O^*(2^n)$ time and $O^*(2^n)$ *i.e.* exponential space.

2.2.2 Interval Sequencing

We are interested in the $1|r_j, \tilde{d}_j|$- problem, described in Example 1.1 p. 28. The set of jobs is $J = \{1, \ldots, n\}$ and an instance is of the form $\mathcal{I} = (p_j, r_j, \tilde{d}_j)_{j \in J}$. There is no objective to minimize, which amounts to minimizing the trivial objective ($\gamma = 0$), which is regular. So, it is sufficient to restrict ourselves to semi-active schedules, defined by the list of their jobs in chronological order. Thus, a schedule S is a permutation $S = (j_1, \ldots, j_n) \in \mathfrak{S}_n$. The completion time of each job j is defined by $\forall j, C_j = \max(C_{j-1}, r_j) + p_j$ with $C_0 = 0$, and a schedule S is admissible *i.e.* is a feasible solution when $\forall j, r_j + p_j \leqslant C_j \leqslant \tilde{d}_j$.

Figure 2.6 shows how to decompose a schedule $S = (j_1, \ldots, j_n)$ into a prefix π and a suffix σ with $S = \pi \cdot \sigma$. We define B as the completion time of the last job of the prefix, and B is also a lower bound of the release times of all the jobs of the suffix. This simple notion actually plays a great role in all scheduling problems.

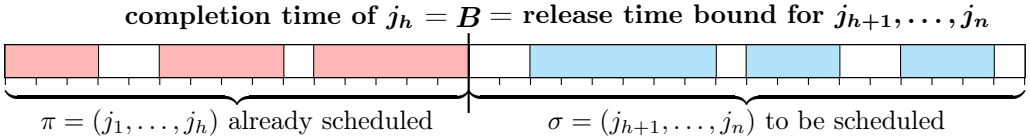


Figure 2.6: A time bound between a schedule prefix and a schedule suffix

Notice that, following Remark 2.5 p. 49, adding a job j to the prefix π by setting $\pi' = \pi \cdot (j)$ amounts to scheduling j and shifting the bound B by setting $B' = C_j = \max(B, r_j) + p_j$. This idea serves both for dynamic programming across subsets and for Inclusion-Exclusion.

Dynamic programming across subsets

For a release time bound B as defined in Figure 2.6 p. 57 and for a job set $X \subset J$, we define $Opt[B, X]$ as the best objective γ of an admissible schedule suffix $\sigma = (j_{h+1}, \ldots, j_n)$ starting at time B and formed of jobs of X, *i.e.* such that $X = \{j_{h+1}, \ldots, j_n\}$. Recall that a minimum taken on an empty set equals $+\infty$ (our minimum is actually an infimum), so, knowing that $\gamma = 0$, $Opt[B, X]$ equals 0 if there is such a schedule and $+\infty$ otherwise.

We have: $\gamma^{opt} = Opt[0, J]$. The expression $Opt[B, X]$ is defined by recurrence, so is computed by dynamic programming: the set $X = \varnothing$ corresponds to the empty schedule suffix, always admissible. For $X \neq \varnothing$, we try all possible jobs j in first position. Their completion time C_j becomes the release bound of the others. We derive:

$$Opt[B, \varnothing] = 0 \tag{2.26}$$

$$Opt[B, X] = \min_{\substack{C_j \leqslant \tilde{d}_j \text{ where} \\ C_j = \max(B, r_j) + p_j}} Opt[C_j, X \setminus \{j\}] \qquad \text{for } X \neq \varnothing \tag{2.27}$$

In the context of this decision problem, one can simplify the way equations are written by setting $Exist[B, X] \iff Opt[B, X] \neq +\infty \iff Opt[B, X] = 0$. The $1|r_j, \tilde{d}_j|$- problem has a solution when $Exist[0, J]$ and we have:

$$Exist[B, \varnothing] = \text{True} \tag{2.28}$$

$$Exist[B, X] = \bigvee_{\substack{C_j \leqslant \tilde{d}_j \text{ where} \\ C_j = \max(B, r_j) + p_j}} Exist[C_j, X \setminus \{j\}] \qquad \text{for } X \neq \varnothing \tag{2.29}$$

To obtain a solution of the problem, we compute the backtrace of the dynamic programming scheme. It is a state graph whose arcs are labeled by jobs, of the form $[B, X] \xleftarrow{j^*} [B', X']$. Each time we compute a minimum in Equation (2.27), we retain a job j^* for which the minimum is reached, *i.e.* $j^* = \arg\min Opt$. Finally, we obtain a (reversed) critical path of the form $[0, J] \xleftarrow{j_1} \cdots \xleftarrow{j_n} [C_{j_n}, \varnothing]$, and we deduce a solution $S = (j_1, \ldots, j_n)$.

Inclusion-Exclusion

The Inclusion-Exclusion method to solve the $1|r_j, \tilde{d}_j|$- problem has been first described by Karp [42], and then enhanced by Nederlof [59].

As shown in section 2.1.1, in order to solve the $\mathcal{P}_{opt} = 1|r_j, \tilde{d}_j|$- problem, it is sufficient to solve the decision problem $\mathcal{P}_{dec}(\mathcal{I}, \pi, \varepsilon)$ defined by Equation (2.1) p. 47. Notice that we only use the value $\varepsilon = 0$ here.

We now implement the $\mathcal{P}_{dec}$ decision problem, as in Definition 2.1 p. 47. Given a release bound B and a job set $J' \subset J$, we count the number $N(B, J')$ of schedules starting from B and formed of jobs of J'. As shown in Figure 2.6 p. 57, imposing a prefix amounts to imposing that the remaining jobs are outside the prefix and have a release time bound B equal to the completion time of the prefix. Formally:

$$\mathcal{P}_{dec}\big(\mathcal{I},\ (\pi_1, \ldots, \pi_h),\ \varepsilon\big) \iff N\big(C_{\pi_h},\ J \setminus \pi\big) > 0 \tag{2.30}$$

The schedules counted by $N(B, J')$ are permutations. We relax the permutation constraint, equivalent to a coverage constraint. The set of relaxed schedules $\mathscr{R}$ is composed of lists of $n' = |J'|$ jobs potentially containing duplicate or missing jobs. It is:

$$\mathscr{R} = \left\{(j_k)_{1 \leqslant k \leqslant n'} \mid \forall\, k, \max(B, C_{j_{k-1}}) + p_{j_k} \leqslant C_{j_k} \leqslant \tilde{d}_{j_k}\right\} \tag{2.31}$$

For a subset of jobs $X \subset J'$, we define $N_X[B, \ell]$ as the number of relaxed schedules of length ℓ starting from B and using only jobs of X, and we derive from Formula (2.17) p. 53:

$$N(B, J') = \sum_{X \subset J'} (-1)^{|J'|-|X|} N_X[B, n'] \tag{2.32}$$

We now just have to compute $N_X[B, \ell]$ by dynamic programming. For $\ell = 0$ there is only one schedule, the empty one, which is admissible. For $\ell > 0$, we try all possible jobs j in the first position. Their completion C_j becomes the release bound of the jobs of the suffix and we recursively count these suffixes. We derive:

$$N_X[B, 0] = 1 \tag{2.33}$$

$$N_X[B, \ell] = \sum_{\substack{j \in X,\ C_j \leqslant \tilde{d}_j \text{ where} \\ C_j = \max(B, r_j) + p_j}} N_X[C_j, \ell - 1] \qquad \text{for } \ell \neq 0 \tag{2.34}$$

As we can see, Formula (2.34) is derived from the same idea as the dynamic programming formula (2.27) p. 58. However, the computation of N is in $O^*(||\mathcal{I}||)$ time and space. Therefore, Inclusion-Exclusion solves the $1|r_j, \tilde{d}_j|$- problem in $O^*(2^n||\mathcal{I}||)$ time and $O^*(||\mathcal{I}||)$ space, *i.e.* pseudopolynomial space, while dynamic programming across subsets solves it in $O^*(2^n)$ time and space, *i.e.* exponential space.

2.3 Related techniques

A lot of scheduling problems are strongly NP-hard permutation problems, for which a brute-force resolution algorithm runs in $O^*(n!)$ time. For these problems, Inclusion-Exclusion can often provide an algorithm running in $O^*(2^n||\mathcal{I}||^{O(1)})$ time and $O^*(||\mathcal{I}||^{O(1)})$ space, as we will show in chapter 4. We review some techniques to improve theoretical and practical complexities.

Because they are of major importance for scheduling problems, we are particularly interested in permutation problems, where the aim is essentially to find in which order the elements of a list must be scheduled.

We reuse all the notations of section 2.1. In particular, we consider a coverage decision problem whose solutions are lists of elements of V with $|V| = n$. The set of relaxed solutions is $\mathscr{R}$ and the set of covering lists is $\mathscr{C}$. Then, the set of covering (*i.e.* strict) solutions is $\mathscr{S} = \mathscr{R} \cap \mathscr{C}$.

In this section, we will frequently have to handle partial sums of binomial coefficients. We recall some useful formulas (see e.g. Graham et al. [33]).

For $\alpha \in]0, 1[$, we define $h(\alpha)$ as the binary entropy of α:

$$h(\alpha) = -\alpha \log_2 \alpha - (1-\alpha) \log_2 (1-\alpha) \tag{2.35}$$

This formula extends by continuity to $h(0) = h(1) = 0$. We have $0 \leqslant h(\alpha) \leqslant 1$, $h(\frac{1}{2}) = 1$ and $\forall\, \alpha, h(1-\alpha) = h(\alpha)$. Moreover, for a single binomial coefficient, we have:

$$\binom{n}{\lfloor \alpha n \rfloor} = \Theta^*\left(2^{h(\alpha)n}\right) \tag{2.36}$$

This result extends to partial sums of binomials. The close-to-middle binomial $\binom{n}{\lfloor n/2 \rfloor}$ is in $\Theta^*(2^n)$, so any sum including this term, *i.e.* any sum $\sum_{k=a(n)}^{b(n)} \binom{n}{k}$ with $a(n) \leqslant \frac{n}{2} \leqslant b(n)$, is also in $\Theta^*(2^n)$.

We now consider a sum involving either the first or the last binomials, excluding the middle term. We have:

$$\sum_{k=0}^{\lfloor \alpha n \rfloor} \binom{n}{k} = \sum_{k=n-\lfloor \alpha n \rfloor}^{n} \binom{n}{k} = \Theta^*\left(2^{h(\alpha)n}\right) \qquad \text{for } \alpha \leqslant \frac{1}{2} \tag{2.37}$$

These formulas still hold when replacing $\lfloor \alpha n \rfloor$ with any expression $f(n)$ close to αn up do an additive constant, *i.e.* $f(n) = \alpha n + O(1)$.

2.3.1 Bonferroni inequalities

These inequalities (see e.g. Galambos and Simonelli [25]) apply on partial sums where index subsets have a limited cardinal k. Depending on the evenness of k, they give an upper or lower bound on the Inclusion-Exclusion sum.

Here is an intuitive presentation of the Bonferroni formula for small values of k. The notation $\sum \text{card}(\bigcap k \text{ sets})$ denotes the sum of the cardinals of the intersections of k sets.

$$\begin{aligned}
\text{card union} &\geqslant 0 && (k=0)\\
&\leqslant \textstyle\sum \text{card}(\bigcap 1 \text{ set}) && (k=1)\\
&\geqslant \textstyle\sum \text{card}(\bigcap 1 \text{ set}) - \sum \text{card}(\bigcap 2 \text{ sets}) && (k=2)\\
&\leqslant \textstyle\sum \text{card}(\bigcap 1 \text{ set}) - \sum \text{card}(\bigcap 2 \text{ sets}) + \sum \text{card}(\bigcap 3 \text{ sets}) && (k=3)
\end{aligned}$$

In the general case, we have these inequalities, to be compared with Equation (2.8) p. 50:

$$\text{card}\Big(\bigcup_{i=1}^{n} A_i\Big) \quad \begin{matrix} k \text{ even} \\ \leqslant \\ \geqslant \\ k \text{ odd} \end{matrix} \quad \sum_{\substack{I \subset \{1,\dots,n\} \\ I \neq \varnothing \\ |I| \leqslant k}} (-1)^{|I|-1} \,\text{card}\Big(\bigcap_{i \in I} A_i\Big) \tag{2.38}$$

Applied to a coverage problem, where $\mathscr{R}$ is the set of relaxed solutions, and $\mathscr{S} = \mathscr{R} \cap \mathscr{C}$ is the set of strict *i.e.* covering solutions, we have these inequalities, to be compared with Equation (2.8) p. 50:

$$\text{card} \underbrace{(\mathscr{R} \cap \mathscr{C})}_{\substack{\text{covering} \\ \text{solutions}}} \quad \begin{matrix} k \text{ even} \\ \leqslant \\ \geqslant \\ k \text{ odd} \end{matrix} \quad \underbrace{\sum_{\substack{X \subset I \\ |X| \geqslant |I|-k}} (-1)^{|I|-|X|} \;\text{card}\; \underbrace{(\mathscr{R} \cap X^*)}_{\substack{\text{relaxed solutions} \\ \text{using only values in } X}}}_{\text{partial Bonferroni sum } S_k} \tag{2.39}$$

We define S_k as the partial Bonferroni sum, right-hand side of inequality (2.39) p. 60. Notice that the smaller k is, the larger the sets X are. In particular, S_0 is the cardinal of the whole set of relaxed solutions, *i.e.* $S_0 = \mathrm{card}\,\mathscr{R}$.

These inequalities may be used to decide whether or not there exists a strict *i.e.* covering solution without computing the 2^n terms of the Inclusion-Exclusion sum. Set $n = |I|$ and let k vary from 0 to n. If k is even and $S_k \leqslant 0$, then $\mathrm{card}(\mathscr{S}) \leqslant 0$ and there is no covering solution. If k is odd and $S_k > 0$, then $\mathrm{card}(\mathscr{S}) > 0$ and there is at least one covering solution. This is implemented in Algorithm 2.7, where Function *ExactBonferroni*() returns *True* when there exists a covering solution. We recall that each $\mathrm{card}(\mathscr{R} \cap X^*)$ is obtained by computing N_X.

Algorithm 2.7: An exact decision procedure using Bonferroni inequalities

Function *ExactBonferroni*():
  $S \leftarrow 0$ // *At each step,* $S = S_k$
  for $k \leftarrow 1, \ldots, n-1$ **do**
    $S \leftarrow S + \sum_{X \subset I, |X| = n-k} N_X$
    if k even **then**
      if $S \leqslant 0$ **then return** False
    else
      if $S > 0$ **then return** True
  $S \leftarrow S + \sum_{X \subset I, |X| = 0} N_X$ // *trivial, computes* $S \leftarrow S_n = S_{n-1} + N_\varnothing$
  if $S > 0$ **then return** True **else return** False

Clearly, Algorithm 2.7 has the same worst-case time complexity as the standard algorithm which computes the whole Inclusion-Exclusion sum. But it can compute less terms, so there is hope for a practical improvement. It has been extensively tested on the $F3||C_{\max}$ problem discussed in chapter 4. Unfortunately, the results do not show any substantial practical improvement compared to the standard algorithm.

We could imagine deriving an approximation scheme from Bonferroni inequalities. For some fixed coefficient $0 < \alpha < \frac{1}{2}$, take $k = 2\left\lfloor \frac{\alpha n}{2} \right\rfloor + 1$, *i.e.* $k \simeq \alpha n$ and k odd. Compute the Bonferroni partial sum S_k. If $S_k > 0$, there is a solution. If $S_k \leqslant 0$, declare, perhaps wrongly, that there is no solution. This is implemented in Algorithm 2.8 p. 62 as Function *ApproxBonferroni*(). This decision procedure is pessimistic and leads to a guaranteed correct but potentially non-optimal solution when it is used to solve a problem by self-reduction, but it can be computed using $O^*(2^{h(\alpha) \times n})$ terms, where $h(\alpha)$ is the binary entropy of α.

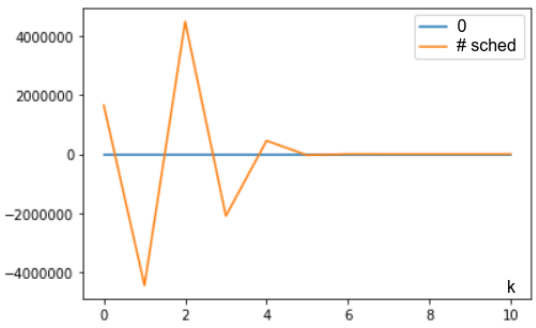
Algorithm 2.8: An approximate decision procedure using Bonferroni inequalities

Function *ApproxBonferroni*():
  $k \leftarrow 2\left\lfloor \frac{\alpha n}{2} \right\rfloor + 1$
  Compute S_k
  if $S_k > 0$ **then**
    return True // *exact*
  else
    return False // *pessimistic*

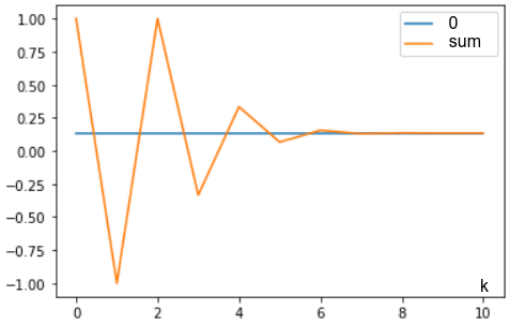
Again, this approximation scheme has been tested on the $F3||C_{\max}$ problem discussed in chapter 4. Unfortunately, the results show unbounded approximation ratios. We actually do not know any permutation problem leading to good approximation ratios with this approach.

To explain the two previous negative conclusions, we can see that alternating sums often lead to numerically very unstable results. For example, the expansion of the exponential gives $e^{-x} = \sum_{i=0}^{+\infty} \frac{(-x)^i}{i!}$, but it is well known that this series converges very badly for $x > 1$.

Figure 2.9 compares the evolution of two partial sums as a function of k. On the left, Figure 2.9a shows the successive partial Bonferroni sums for a small but representative instance of the $F3||C_{\max}$ problem, with $n = 10$ jobs. On the right, Figure 2.9b shows successive partial sums $\sum_{i=0}^{k} \frac{(-2)^i}{i!}$ approaching e^{-2}.



(a) Bonferroni partial sums



(b) $\sum_{i=0}^{k} \frac{(-2)^i}{i!} \simeq e^{-2}$ partial sums

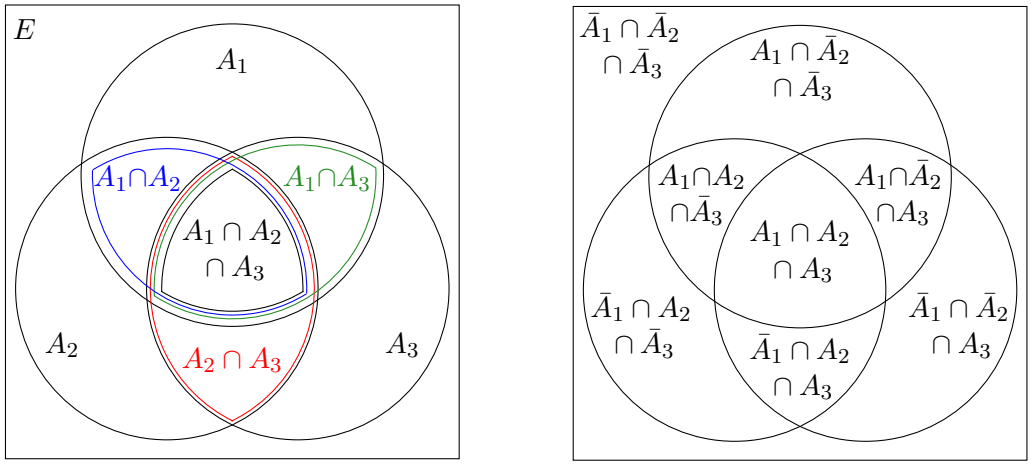
Figure 2.9: Comparison between Bonferroni and partial sums of a negative exponential.

As we can see, both diagrams look alike. In both cases, the alternating sign $(-1)^n$ makes the limit of the sum very small compared to the amplitude of the oscillations, thus it is very difficult to accurately estimate the limit without computing many terms.

There are other techniques that can potentially exploit Bonferroni inequalities, in particular Abstract Tubes, described in section 2.3.3.

2.3.2 Möbius inversion

This technique (see Mazur [53], Nederlof [60]) is a way of simplifying Inclusion-Exclusion formulas. As an introductory example, Figure 2.10 shows the Venn diagramm of 3 sets A_1, A_2, A_3 in a universe E. In its traditional form (on the left), the Venn diagram represents the $2^3 = 8$ partial intersections, which correspond to the interior of a closed curve: a square, 3 circles, 3 lenses, and a curved triangle. We can also (on the right) partition the Venn diagram into $2^3 = 8$ regions, which are disjoint from eachother and whose union form the universe E. The idea of Möbius inversion is to express an Inclusion-Exclusion sum using the regions and no longer the sets.



(a) Sets (b) Regions

Figure 2.10: A Venn Diagram composed of three sets and partitioned into regions

Consider the Venn diagram associated with the set family $(A_1, \ldots, A_n)$ with $A_i \subset E$. It contains $(2^n - 1)$ regions (excluding the outside of the union). Each region is the intersection of each A_i or its complementary $\bar{A}_i$. To each index set $R \subset I$ we associate the region $E_R = \bigcap_{i\in R} A_i \cap \bigcap_{i\notin R} \bar{A}_i = \bigcap_{i\in R} A_i \setminus \bigcup_{i\in F} A_i$ where $F = I \setminus R$. Elements of A_i for $i \in R$ are required, elements of A_i for $i \in F$ are forbidden.

Notice that some regions may be empty. Let us consider only index sets R corresponding to non-empty regions E_R, and let us enumerate them in decreasing inclusion order: we get a suite $R_1 \ldots R_K$ with $K < 2^n$ and $\forall\, i, j, R_i \supset R_j \Rightarrow i \leqslant j$. We define the ζ-matrix Z of size K by taking $\forall\, i, j \leqslant K, Z_{ij} = 1$ if $R_i \supset R_j$, 0 otherwise. This matrix has a determinant equal to 1, it is invertible and its inverse Z^{-1} has integer coefficients. Let us state (Möbius inversion): $\begin{pmatrix}\alpha_1\\ \cdots\\ \alpha_K\end{pmatrix} = Z^{-1}\begin{pmatrix}1\\ \cdots\\ 1\end{pmatrix}$, with values α_k being potentially exponential in n. Then, we have:

$$\operatorname{card} \bigcup_{i\in I} A_i = \sum_{k=1}^{K} \alpha_k \operatorname{card} \bigcap_{i\in R_k} A_i. \tag{2.40}$$

Goaoc et al. [31] described further enhancements of this method.

Formula (2.40) is computationally interesting only if the number of non-empty regions of the Venn diagram is exponentially smaller than 2^n. We did statistics on the $F3||C_{\max}$ problem discussed in chapter 4. On typical instances, the proportion of non-empty regions

is close to 50 %, *i.e.* the number of non-empty regions is close to 50 % $\times 2^n$, so in this case this technique leads to no improvement. We don't know any permutation problem for which this technique significantly improves the worst-case time complexity bound.

Venn diagram regions and branching

Venn diagram regions are useful, not only to derive simplified Inclusion-Exclusion formulas, but also to derive branching schemes. This technique has been introduced by Bax [6] and considerably developed by Nederlof et al. [62].

The idea is to consider index sets $R \subset I$ of required properties and index sets $F \subset I$ of forbidden properties. Together, they correspond to the set $\bigcap_{i \in R} A_i \setminus \bigcup_{i \in F} A_i$. The regions in the Venn diagram correspond to the case where $F = I \setminus R$, but we can consider any couple (R, F) such that $R \cap F = \varnothing$ without imposing $R \cup F = I$. The index set $X = I \setminus (R \cup F)$ corresponds to optional properties: neither required, nor forbidden. This way, one can deduce recurrence formulas usable in a branch-and-reduce algorithm, whose complexity may be analyzed by measure-and-conquer (see e.g. Fomin and Kratsch [23]).

This technique fruitfully applies to numerous graph algorithms, in which solutions are expressed as sets and data order in solutions has no importance. But it seems difficult to manage required properties inside a polynomial counting scheme for a permutation problem, as it is impossible to impose an ordering constraint between elements (this ordering is indeed the result to compute).

2.3.3 Abstract Tubes

We refer to Dohmen [21], Naiman and Wynn [56], Narushima [57, 58]. The theory of abstract tubes aims at producing simplified but exact Inclusion-Exclusion formulas. It relies on algebraic topology, where a multigraph is identified with a solid in a multidimensional space. According to the algebraic topology notations, the solid is called a concrete simplicial complex and the multigraph is called an abstract simplicial complex.

We first recall the definition of a multigraph, which is a generalization of a graph. A multigraph is built upon a set of indices I. Each singleton $\{i\}$ for $i \in I$ is identified to a vertex, each pair of indices $\{i \neq i'\}$ is identified to an edge, and in general, each index set $F \subset I$ is identified to a generalized edge, called a face. So, formally, a multigraph is a set of faces $\mathscr{F} \subset \mathscr{P}(I)$. Moreover, in a graph, the extremities of all edges must be vertices. The same way, the set of faces $\mathscr{F}$ of a multigraph is supposed to be stable by inclusion: if $F' \subset F$ and $F \in \mathscr{F}$ then $F' \in \mathscr{F}$.

To establish a correspondence between an abstract simplicial complex, *i.e.* a multigraph, and a concrete simplicial complex, *i.e.* a solid, we define a multidimensional affine space S and we assign a point $P_i \in S$ to each index $i \in I$. Then, each abstract face $F \subset I$ corresponds to the convex hull of the points $\{P_i, i \in F\}$, called a simplex, and the multigraph corresponds to the solid formed by the union of simplices, hence the term of simplicial complex. When the concrete solid associated with the abstract multigraph is connected and without hole, the abstract multigraph is said to be contractible.

We now consider a family $(A_i)_{i \in I}$ of subsets of a universe E, and an abstract simplicial complex, *i.e.* a set of faces $\mathscr{F} \subset \mathscr{P}(I)$, stable by inclusion, whose set of vertices is precisely derived from I, *i.e.* $\{i \mid \{i\} \in \mathscr{F}\} = I$.

For each element $x \in E$ of the universe, we define the subcomplex $\mathscr{F}_x$ formed by the faces *i.e.* index sets F corresponding to partial intersections of A_i which contain x. Formally:

$$\mathscr{F}_x = \{F \in \mathscr{F} \mid x \in \bigcap_{i \in F} A_i\} \tag{2.41}$$

Then, by definition, $\mathscr{F}$ is said to be an abstract tube (with respect to $(A_i)_{i \in I} \subset E$) when each $\mathscr{F}_x$ for $x \in E$ is contractible.

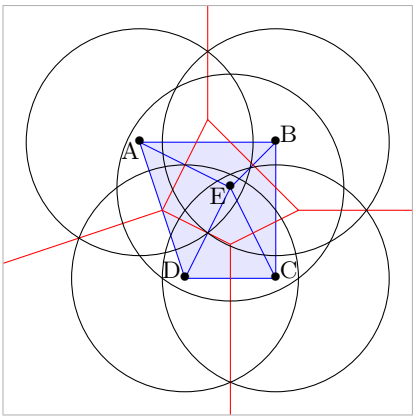
The main interest of abstract tubes is due to this relation: if $\mathscr{F}$ is an abstract tube with respect to $(A_i)_{i \in I} \subset E$, then the following equality holds:

$$\operatorname{card}\Big(\bigcup_{i \in I} A_i\Big) = \sum_{F \in \mathscr{F}} (-1)^{|F|-1} \operatorname{card}\Big(\bigcap_{i \in F} A_i\Big) \tag{2.42}$$

As a consequence, $\operatorname{card}\big(\bigcup_{i \in I} A_i\big)$ can be computed as a sum with $|\mathscr{F}|$ terms instead of 2^n. Notice that Bonferroni inequalities also hold in an abstract tube. The abstract simplicial complex $\mathscr{F} = \mathscr{P}(I)$ is a trivial abstract tube for which formula (2.42) reduces to the usual Inclusion-Exclusion sum. In general, formula (2.42) is interesting only in the case of an abstract tube with a small number of faces.

The most spectacular result is due to Naiman and Wynn [56], who show how to compute the volume of the union of balls in a d-dimension Euclidean space in $O\big(\sum_{k=1}^{d} \binom{n}{k}\big)$ terms. This means $O(n^d)$ terms if d is independent of n, or $O^*(2^{h(\alpha)n})$ terms if $d \sim \alpha n$, where $0 < \alpha < 1$ and $h(\alpha)$ is the binary entropy of α. This result is to be compared with the $O(2^n)$ terms of the traditional Inclusion-Exclusion formula.

Naiman and Wynn use an abstract tube based on the Delaunay triangulation. Figure 2.11 shows their example for 5 disks A, B, C, D, E on a plane (d=2). The surface measure (μ) replaces the cardinal (card). They achieve a 17-term sum whereas there are 2^5=32 non-empty partial intersections.



Voronoï diagram
Delaunay complex

- 5 vertices: $\{A\}, \{B\}, \{C\}, \{D\}, \{E\}$.
- 8 edges: $\{AB\}, \{BC\}, \{CD\}, \{DA\}, \{AE\}, \{BE\}, \{CE\}, \{DE\}$.
- 4 faces: $\{ABE\}, \{BCE\}, \{CDE\}, \{DAE\}$.

$$\begin{aligned}
& \mu(A \cup B \cup C \cup D \cup E) \\
= \; & \mu(A) + \mu(B) + \mu(C) + \mu(D) + \mu(E) \\
- \; & \mu(A \cap B) - \mu(B \cap C) - \mu(C \cap D) - \mu(D \cap A) \\
- \; & \mu(A \cap E) - \mu(B \cap E) - \mu(C \cap E) - \mu(D \cap E) \\
+ \; & \mu(ABE) + \mu(BCE) + \mu(CDE) + \mu(DAE)
\end{aligned}$$

Figure 2.11: Surface of a union of disks using a Voronoï/Delaunay based abstract tube

We can use a discrete measure μ concentrated at some points, so that the measure of a disk is the number of points inside it instead of its surface. But an Euclidean distance is derived from a norm with a strict form: from a computational point of view, we must have $||(x_1, \ldots, x_d)||^2 = \sum_i (\sum_k M_{ik} x_k)^2$ where M is an invertible matrix. So, unfortunately, relating this result to a scheduling or operational research problem turns to be very difficult.

The most promising abstract tubes are produced by Dohmen's theorems [21, pp. 28, 34]. Unfortunately, they are valid only under the assumption that, for many index subsets R, $\bigcap_{i\in R} A_i \subset \bigcup_{i\notin R} A_i \iff \bigcap_{i\in R} A_i \cap \bigcap_{i\notin R} \bar{A}_i = \varnothing \iff E_R = \varnothing$, where E_R is the Venn diagram region defined in section 2.3.2. So, these theorems suppose that many Venn diagram regions are empty. Hence, we may expect abstract tubes based formulas to be not significantly shorter than Möbius inversion based ones.

To the best of our knowledge, although the theory of abstract tubes is often perceived as being promising, there is no concrete application to operational research in the literature.

2.3.4 Index space trimming

This technique consists in cleverly choosing enumerations such that index sets X outside some well defined and easily iterable index set collection are mathematically guaranteed to produce null terms. For example, if we exhibit two subsets $X_1 \subset X_2 \subset I$ such that for all $X \subset I$, $\operatorname{card}\bigcap_{i\in X} A_i = 0$ unless $X_1 \subset X \subset X_2$, then we derive a formula with $2^{|X_2|-|X_1|}$ terms. This way, Nederlof [61, p. 881] counts the number of perfect matchings in a n-vertex graph in $O^*\big(2^{\frac{1}{2}(h(\frac{1}{3})+1)n}\big) \simeq O^*\big(2^{0.9591n}\big) \simeq O^*(1.945^n)$ time instead of $O^*(2^n)$.

Once again, it happens that the technique we are going to expose cannot apply to permutation problems, because there is a need to impose an order in the relaxed lists, whereas in a permutation problem this order is the result we are looking for. But it's worth describing the algorithm by Nederlof, not only because it illustrates the index trimming technique, but also because it is an anthology of counting techniques around Inclusion-Exclusion.

We consider an undirected graph $G = (V, E)$ with $E \subset \mathscr{P}_2(V)$, where $\mathscr{P}_2$ denotes the set of parts with exactly 2 elements. We also define, for $V' \subset V$, $G[V']$ as the subgraph of G composed of the vertices of V' and the corresponding edges, and for a vertex v, $d_G(v)$ as the degree of v in G (0 if v is not a vertex of G). For $A \subset V$ and $B \subset V$, an A–B edge is an edge whose endpoint is in A and the other in B.

The problem is stated as follows: count the number of perfect matchings of G. A perfect matching $M \subset E$ is a set of edges which covers all vertices, and in which 2 distinct edges do not have a common vertex. In other words, if $|V| = n$, a perfect matching M of G is a partition of V of the form $V = \{u_1, v_1\} \uplus \ldots \uplus \{u_p, v_p\}$, with $\forall\, i, \{u_i, v_i\} \in E$, where $\uplus$ is the union of disjoint sets. Obviously this implies that n is even, with $n = 2p$.

To determine the number n_M of perfect matches, the straightforward application of the Inclusion-Exclusion formula consists in counting in $(V^2)^p$ the covering lists of the form $((u_1, v_1), \ldots (u_p, v_p))$ with $\forall\, i, \{u_i, v_i\} \in E$. A relaxed list is made up of p independent pairs (u_i, v_i). There are $p!$ possible orders for pairs and 2 orders per pair. So, applying Formula (2.16) p. 53 without trying to simplify it, we obtain:

$$n_M = \frac{1}{p!2^p} \sum_{X\subset V} (-1)^{2p-|X|} \left(\frac{\underbrace{\operatorname{card}\{(u,v) \in X^2 \mid \{u,v\} \in E\}}}{2\operatorname{card}(E \cap \mathscr{P}_2(X))} \right)^p \tag{2.43}$$

This formula can be computed in $O^*(2^{2p}) = O^*(2^n)$ steps.

Here is now the method used by Nederlof to get a better complexity: let us arbitrarily partition V into 2 subsets of same size: $V = A \uplus B$ with $|A| = |B| = p$, and consider a perfect matching M, as in Figure 2.12 p. 67. We define the "left bridge side" $L(M)$ as the

set of extremities in A of the edges A–B, *i.e.* $L(M) = \{a \mid \{a,b\} \in M \wedge b \in B\}$. We define $\ell(M)$ as the number of edges A–B, $q(M)$ as the number of edges A–A (equal to the number of edges B–B), and $k(M)$ as the number of edges that hit A, *i.e.* the edges A–A or A–B. We get: $\ell(M) = |L(M)| \leqslant p$, $q(M) = \frac{p+\ell(M)}{2}$, $k(M) = q(M) + \ell(M) = \frac{p+\ell(M)}{2}$, p and $\ell(M)$ have same parity. Whenever there is no ambiguity we will write ℓ, q, k instead of $\ell(M), q(M), k(M)$.

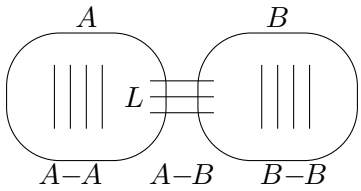


Figure 2.12: Edges of a perfect matching with set vertex partitioned into 2 halves

The method by Nederlof consists in computing, ℓ by ℓ, the number n_ℓ of matchings M such that $\ell(M) = \ell$, and adding them. We can compute n_ℓ by two auxiliary algorithms: one in $O^*\big(\binom{p}{\ell}2^p\big)$ and one in $O^*\big(\binom{p}{k}2^p\big)$. We choose the best algorithm for each ℓ, the worst case is when $\ell(M) \simeq k(M) \simeq \frac{p}{3}$, which finally gives, according to Formula (2.36) p. 60, a complexity in $O^*\big(\binom{p}{p/3}2^p\big) = O^*\big(2^{(h(\frac{1}{3})+1)p}\big) = O^*\big(2^{\frac{1}{2}(h(\frac{1}{3})+1)n}\big)$.

Computation of n_ℓ in $O^*\big(\binom{p}{\ell}2^p\big)$ time

We express n_ℓ as $n_\ell = \sum_{|L|=\ell} n_L$ and we calculate each n_L. For this computation, we present a different (and simplified) version of the one by Nederlof. As shown in Figure 2.12, a perfect matching M with left bridge side L is the concatenation of a perfect matching M_A of $G[A \setminus L]$ with a perfect matching M_B of $G[L \cup B]$ of which each vertex of L is the extremity of a unique edge. We define n_A as the number of possible M_A and n_B as the number of possible M_B. We therefore have $n_L = n_A \times n_B$, and n_A is computed in $O^*(2^{p-\ell})$ by Formula (2.43) p. 66.

It remains to compute n_B. We index the elements of L by writing $L = \{L_1, \ldots, L_\ell\}$. Computing n_B amounts to counting the covering lists in B of the form $((x_1, \ldots, x_\ell), (u_1, v_1), \ldots, (u_q, v_q))$, where each $\{L_i, x_i\}$ and each $\{u_j, v_j\}$ is an edge. Taking into account the possible orders for the q pairs, we have:

$$n_B = \frac{1}{q!2^q} \sum_{X \subset B} (-1)^{p-|X|} \prod_{i=1}^{\ell} \underbrace{\operatorname{card}\left\{ \begin{matrix} x \in X \mid \\ \{L_i, x\} \in E \} \end{matrix} \right\}}_{d_{G[L \cup X]}(L_i)} \times \Big(2 \operatorname{card}(E \cap \mathscr{P}_2(X))\Big)^q \quad (2.44)$$

This formula is computed in $O^*(2^p)$, so $n_L = n_A \times n_B$ is computed in $O^*(2^p)$. Finally, $n_\ell = \sum_{|L|=\ell} n_L$ is a sum of $\binom{p}{\ell}$ terms, and it is computed in $O^*(\binom{p}{\ell}2^p)$.

Computation of n_ℓ in $O^*\big(\binom{p}{k}2^p\big)$ time

In this computation the order of the elements plays an important role. We order the elements of A arbitrarily by writing $A = a_1 < \ldots < a_p$. We represent a perfect matching M by a list containing on the left the edges $\{x_i, y_i\}$ hitting A, therefore of the form A–A

and A–B, and on the right the edges $\{u_i, v_i\}$ of the form B–B. In addition, for the purposes of the algorithm, we impose $x_1 < \ldots < x_k$. We therefore want to count the lists covering V of the form $((x_1, y_1), \ldots, (x_k, y_k), (u_1, v_1), \ldots, (u_q, v_q))$, with $x_i \in A$, $y_i \in A \cup B$, $u_i \in B$, $y_i \in B$, $\{x_i, y_i\} \in E$, $\{u_j, v_j\} \in E$. We recognize the concatenation of (x_i, y_i) and (u_j, v_j). By taking into account the two possible orders per couple for the $2q$ edges A–A and B–B and the $q!$ possible enumeration orders of edges B–B:

$$n_\ell = \frac{1}{q! 2^{2q}} \sum_{\substack{X \subset V \\ |X \cap A| \geqslant k}} (-1)^{2p-|X|} \; N_X \; \times \; \Big(2 \operatorname{card}(E \cap \mathscr{P}_2(X))\Big)^q \tag{2.45}$$

where N_X is defined as the number of lists of the form $((x_1, y_1), \ldots, (x_k, y_k))$ with $\forall\, i$, $y_i \in X$, $x_i \in A \cap X$, $\{x_i, y_i\} \in E$ and $x_1 < \ldots < x_k$. This is where we use the index trimming technique: the property $x_1 < \ldots < x_k$ is unfeasible if $A \cap X$ does not contain at least k elements. By decomposing $X = (X \cap A) \uplus (X \cap B)$, and knowing that $k \geqslant \frac{p}{2}$, there are, following Formula (2.36) p. 60, $\sum_{k' \geqslant k} \binom{p}{k'} = O^*\big(\binom{p}{k}\big)$ possibilities for $(X \cap A)$ and 2^p possibilities for $(X \cap B)$, so $O^*\big(\binom{p}{k} 2^p\big)$ terms to compute in the sum.

It remains to show how to compute N_X in polynomial time. We have $N_X = n_X[p, k]$ where $n_X[r, m]$ is the number of lists with m pairs of the form $((x_1, y_1), \ldots, (x_m, y_m))$ with $\forall\, i$, $y_i \in X$, $x_i \in A \cap X$, $\{x_i, y_i\} \in E$, $x_1 < \ldots < x_m$, and in addition $x_m \leqslant a_r$. And n_X is calculated by dynamic programming in at most $pk = O^*(1)$ states, as described below.

If $m > r$, we cannot have $x_m \leqslant a_r$, so no sequence is admissible and $n_X[r, m] = 0$. If $m \leqslant 0$, there is a unique empty sequence, so $n_X[r, 0] = 1$. Otherwise, consider the last element x_m, which is also the biggest. Let $x_m < a_r$, and then $x_m \leqslant a_{r-1}$, there are $n_X[r-1, m]$ possibilities for $((x_1, y_1), \ldots, (x_m, y_m))$. Let $x_m = a_r$, and then $u_{m-1} \leqslant a_{r-1}$, there are $n_X[r-1, m-1]$ possibilities for $((x_1, y_1), \ldots, (x_{m-1}, y_{m-1}))$ to be multiplied by the number of pairs $\{x_m, y_m\} = \{a_r, y_m\} \in E$ which equals $d_{G[X]}(a_r)$. Finally, we get:

$$n_X[r, m] = 0 \qquad \text{if } m > r \tag{2.46}$$

$$n_X[r, m] = 1 \qquad \text{if } m \leqslant 0 \tag{2.47}$$

$$n_X[r, m] = n_X[r-1, m] + n_X[r-1, m-1] \times d_{G[X]}(a_r) \qquad \text{otherwise} \tag{2.48}$$

This completes the description of the algorithm by Nederlof.

2.3.5 Zero sweeping

As we saw in the examples of section 2.2, we frequently compute an Inclusion-Exclusion sum whose terms are themselves calculated by dynamic programming. While the index trimming technique consists in detecting in advance, by mathematical analysis, that certain terms of a sum of Inclusion-Exclusion are necessarily zero, the zero sweeping technique consists in detecting null terms on the fly and sweeping them out. It is in fact a question of detecting as soon as possible that a whole class of terms is zero and to avoid calculating them.

The Zero Sweeping Technique mixes two approaches: on the one hand avoid brute-force enumeration of the 2^n subsets used as summation index; on the other hand exploit the fact that the backtrace graphs of dynamic programming are nested within each other to compute them incrementally. While the former technique takes up the ideas of Bax and Nederlof, the latter technique does not appear, to the best of our knowledge, anywhere explicitly in the literature, so this is one of our contributions.

A recursion scheme for Inclusion-Exclusion sums

Following Equations (2.16) p. 53 and (2.17) p. 53, given a set V of elementary values and a set $\mathscr{R}$ of relaxed admissible lists, the aim is to compute the Inclusion-Exclusion sum $S = \sum_{X \subset V} (-1)^{|V|-|X|} N_X$ where $N_X = \text{card}(\mathscr{R} \cap X^*)$. So, N_X is a non-negative and non-decreasing function of X. Thus, if $N_X = 0$, then $\forall\, X' \subsetneq X, N_{X'} = 0$. These numbers do not have to be computed, which saves up to $(2^{|X|}-1)$ computations.

In the spirit of Bax and Nederlof branching studied in section 2.3.2, but without considering any set of required jobs, which are hard to count with permutation problems, we now define job intervals $V_k = \{1 \dots k\}$, and the partial Inclusion-Exclusion sum $P(k, Y)$, for each $k \leqslant n$ and Y such that $V_k \cap Y = \varnothing$:

$$P(k, Y) = \sum_{Y \,\subset\, X \,\subset\, V_k \cup Y} (-1)^{|V_k \cup Y| - |X|} N_X \tag{2.49}$$

We have $S = P(n, \varnothing)$, and we derive the following recurrence equations:

$$P(0, Y) = N_Y \tag{2.50}$$

$$P(k, Y) = P(k{-}1, Y \cup \{k\}) - P(k{-}1, Y) \qquad \text{for } k > 0 \tag{2.51}$$

We introduce (Algorithm 2.13) an extra parameter N with the invariant that $N = N_{V_k \cup Y}$. Notice that any N_X for $Y \subset X \subset V_k \cup Y$ and thus $P(k, Y)$ is null as soon as $N_{V_k \cup Y}$ is null. Finally, we have $S = P(n, \{\}, N_V)$.

Algorithm 2.13: Recursive computation of an Inclusion-Exclusion sum

```
Function P(k, Y, N):          // invariants: V_k ∩ Y = ∅, N = N_{V_k ∪ Y}
    if N = 0 then
        return 0              // saves 2^k − 1 computations
    else if k = 0 then
        return N
    else
        return P(k−1, Y ∪ {k}, N) − P(k−1, Y, N_{V_{k−1} ∪ Y})
```

Incremental dynamic programming state space reduction

The previous technique can be combined with incremental reduction of the dynamic programming state space. We denote by s a dynamic programming state. The initial (root) state r corresponds to $N_X = N_X[r]$, and recursive equations follow this scheme, in which *state′* and *condition* are to be replaced according to the problem to be solved:

$$N_X[s] = 1 \qquad \text{if } s \text{ is a final state} \tag{2.52}$$

$$N_X[s] = \sum_{v \in X \;\mid\; condition(s,v)} N_X[state'(s, v)] \qquad \text{otherwise} \tag{2.53}$$

Let us define G_X as the backtrace of the dynamic programming scheme, *i.e.* as the graph of recursive calls of function N_X. It is an n-ary tree with node sharing. Nodes are states, the root of the tree is r, and links are labeled by values $v \in V$: we denote them by $s \xleftarrow{v} s'$

and they are derived from Equation (2.53) p. 69. Notice that, given G_X and $v \in X$, it is easy to derive $G_{X\setminus\{v\}}$: withdraw links $s \xleftarrow{v} s'$ and keep states which can reach root. We denote by $G \setminus \{v\}$ the graph obtained by removing v, so $G_X \setminus \{v\} = G_{X\setminus\{v\}}$.

To compute N_X, we can proceed in two phases: first build G_X, and then apply dynamic programming itself by traversal of G_X (with caching): $N_X[s] = \sum_{s' \mid s \to s'} N_X[s']$. During this second phase, we can safely remove states s such that $N_X[s] = 0$ and derive a simplified but correct graph G'_X. Notice that the simplification benefits to the $(2^{|X|}-1)$ computations of all $N_{X' \subset X}$ since $\forall\, s, N_X[s]{=}0 \Longrightarrow N_{X'}[s]{=}0$, and $G'_{X'} \subset G'_X$.

We call *dynprog* the function which returns the simplified graph and the result of dynamic programming, *i.e.* $dynprog(G_X)$ computes the couple (G'_X, N_X). We adapt the function P in Algorithm 2.14, and, as a result, the number S of covering admissible lists is given by $S = P(n, \{\}, dynprog(G_V))$.

Algorithm 2.14: Inclusion-Exclusion sum with incremental dynamic programming state space reduction

```
Function P(k, Y, (G', N)):        // invariants: V_k∩Y=∅, G'=G'_{V_k∪Y}, N=N_{V_k∪Y}
    if N = 0 then
        return 0                  // saves 2^k − 1 computations
    else if k = 0 then
        return N
    else
        return P(k−1, Y ∪ {k}, (G', N)) − P(k−1, Y, dynprog(G' \ {k}))
                                  // dynprog withdraws many states
```

This technique multiplies memory consumption by n, but it can improve the actual running time and it cannot degrade the worst-case time complexity bound. From a practical point of view, it has been tested on the $F3||C_{\max}$ problem, discussed in chapter 4, whose worst-case time complexity bound is in $O^*(2^n||\mathcal{I}||)$. The results show a significant speed improvement by a constant factor due to incremental dynamic programming, but unfortunately they show no improvement in the number of terms of the Inclusion-Exclusion sum, leading to the same worst-case complexity. We do not know any permutation problem for which it is improved.

2.4 Conclusions

In this chapter, we described the principles of Inclusion-Exclusion, and we compared this technique with dynamic programming across subsets. Both these techniques exploit the same idea: the self-reducibility of a problem. Starting from the same recurrence equations describing the effect of appending an element to a prefix, we can derive both a dynamic programming scheme across subsets and an Inclusion-Exclusion process. But Inclusion-Exclusion has a pseudopolynomial worst-case space complexity. From a theoretical point of view, this is a major improvement over dynamic programming across subsets, which has an exponential worst-case space complexity.

Using dynamic programming across subsets, Jansen et al. [40] showed that a wide class of scheduling problems with n jobs can be solved in $O^*(c^n||\mathcal{I}||^{O(1)})$ time and $O^*(c^n||\mathcal{I}||^{O(1)})$ space, for some constant c. By combining Inclusion-Exclusion and dynamic programming on relaxed schedules, we can solve several of these problems in $O^*(2^n||\mathcal{I}||^{O(1)})$ time and $O^*(||\mathcal{I}||^{O(1)})$ space, notably parallel machine problems in Chapter 3 and permutation flow-shop problems in Chapter 4.

In the context of permutation problems, where the aim is essentially to determine in which order elements have to be placed, we reviewed in the literature several techniques to accelerate Inclusion-Exclusion: Bonferroni inequalities, Möbius inversion, Abstract Tubes, and Index space trimming. We also contributed to a new one, called Zero Sweeping.

From a theoretical point of view, in the usual case where each counting of relaxed solutions has a pseudopolynomial time complexity. the standard application of Inclusion-Exclusion achieves a proven worst-case time complexity in $O^*(2^n||\mathcal{I}||^{O(1)})$, with an exponential growing rate of 2, It seems rather difficult to improve this growing rate. All improvements introduced by Nederlof [61] exploit ordering constraints incompatible with permutation problems.

From a practical point of view, the standard application of Inclusion-Exclusion is a quite brute-force method requiring computation of an exponential number of terms. The techniques we reviewed do not exponentially improve the practical computation time. In Chapter 5 we attempt another approach: use an Inclusion-Exclusion based Lagrangian relaxation.

Contributions

Beyond presenting the Inclusion-Exclusion approach, we have reviewed and experimented techniques described in the literature to accelerate Inclusion-Exclusion computations. Among these techniques, we have two original contributions:

- Exploit Bonferroni inequalities to derive an approximation scheme.
- Save calculations by exploiting the fact that, during the computation of the terms of the sum of Inclusion-Exclusion, dynamic programming backtrace graphs are nested inside each other.

These studies produced negative results, which explains why they have not been published.

Chapter 3

Parallel machine scheduling

Topics

- We deal with unrelated parallel machine scheduling problems in presence of job release dates and deadlines, to minimize general regular maximum and sum objective functions.
- Using Inclusion-Exclusion, we provide a generic algorithm for solving these problems, while running with moderate exponential-time and pseudopolynomial-space worst-case complexities.
- We emphasize the importance of reducing a relaxed problem with m parallel machines to m independent single-machine relaxed problems, in order to reduce complexities.
- This reduction is quite classical for a maximum-type objective, but more elaborate for a sum-type objective, requiring to use generating series and convolution.
- We also emphasize the importance of adapting dynamic programming schemes to cope with large cardinals involved in Inclusion-Exclusion, in order to reduce complexities.

In this chapter we are interested in scheduling a set of jobs on unrelated parallel machines in the presence of job release dates and deadlines, while minimizing general regular maximum or total cost objective functions. Using Graham's notations, these problems are denoted by $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ or $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$. They have been described in section 1.1.1 but we hereafter recall the precise notations we will use throughout this chapter.

In the $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ and $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ problems, there are n jobs to be scheduled on m unrelated machines, *i.e.* machines with different features. Each job can be processed by any machine, but each machine can process only one job at a time. When scheduled on machine i, job j has a processing time p_{ij}, cannot start before its release time r_{ij}, cannot complete after its deadline $\tilde{d}_{ij}$, and is associated with an individual cost function f_{ij}. The individual cost functions f_{ij} that we consider are regular, *i.e.* non-decreasing

with respect to the completion time they take as argument. For any given schedule S, we define $C_j(S)$ as the completion time of j and $I_j(S)$ as the machine assigned to j in schedule S. Then, the objective function is defined as $\gamma(S) = \max_{1 \leqslant j \leqslant n} \left(f_{I_j(S),j}(C_j(S))\right)$ or $\gamma(S) = \sum_{1 \leqslant j \leqslant n} \left(f_{I_j(S),j}(C_j(S))\right)$. The aim is to find an optimal solution which minimizes the objective function $\gamma(S)$. Whenever there is no ambiguity we will drop off the mention to S in the notations.

We define $J = \{1, \ldots, n\}$ as the set of jobs to be scheduled. An instance $\mathcal{I}$ of the $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ or $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ problem is of the form $\mathcal{I} = (p_{ij}, r_{ij}, \tilde{d}_{ij}, f_{ij})_{1 \leqslant i \leqslant m,\, j \in J}$.

While particular cases of the $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ and $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ problems have been extensively studied, as reviewed in section 1.3, few attempts have been done to cope with the general case. Fomin and Kratsch [23] describe an algorithm for the single machine problems $1||f_{\max}$ and $1||\sum f_j$ with an $O^*(2^n)$ time and space complexity. Lenté et al. [50] describe an algorithm for the parallel machine problems $P||f_{\max}$ and $P||\sum f_j$ with an $O^*(3^n)$ time complexity and an $O^*(2^n)$ space complexity. Jansen et al. [40] describe a very general algorithm class, with an $O^*(2^{O(n)}||\mathcal{I}||^{O(1)}) = O^*(c^n||\mathcal{I}||^{O(1)})$ time and space complexity for some c.

Besides, under the Exponential Time Hypothesis introduced by Impagliazzo and Paturi [38], Jansen et al. [40] showed that the two sub-problems denoted by $P2||C_{\max}$ and $P||\sum w_j C_j$ do not have sub-exponential algorithms, thus justifying the design of exact exponential algorithms, including Inclusion-Exclusion based ones, for the $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ and $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ problems.

Our main contribution is to provide a generic algorithm to minimize $\gamma = f_{\max}$ and $\gamma = \sum f_{ij}$ objective functions, while running with moderate exponential-time and pseudopolynomial-space worst-case complexities. Let γ^{opt} be the optimum objective value, for any instance $\mathcal{I}$. Then, our algorithm solves the $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem in $O^*(2^n||\mathcal{I}||)$ time and $O^*(||\mathcal{I}||)$ space, and the $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ problem in $O^*(2^n||\mathcal{I}||\gamma^{opt})$ time and $O^*(||\mathcal{I}||\gamma^{opt})$ space.

The algorithm we describe for the $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem follows up but generalizes Karp's ideas [42]. Actually, as developed in Remark 3.3 p. 82, specialization of our algorithm to the $P||C_{\max}$ problem and the $1|r_j, \tilde{d}_j|$- problem ($1|r_j, \tilde{d}_j|f_{\max}$ with the trivial cost $f_j = 0$) leads to a close but enhanced version of Karp's one. In his paper, Karp considers that operations on cardinals take constant time, which seems optimistic because cardinals can be rather large. If we consider that arithmetic operations take quasi-linear time with respect to their result size in bits, computing on cardinals induces a hidden complexity of $O^*(||\mathcal{I}||)$. Using an equivalent yet more precise counting scheme, we reduce this hidden complexity to $O^*(1)$.

In section 3.1, we describe how to apply the Inclusion-Exclusion principle to unrelated parallel machine scheduling problems, and how to transform a relaxed problem with m parallel machines into m independent single-machine relaxed problems. These principles are illustrated through the fundamental example of the $P||C_{\max}$ problem.

In sections 3.2 p. 80 and 3.3 p. 83 we apply these principles to the $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ and $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ problems. As we will see in this chapter, but also in chapter 4, we can distinguish two radically different phases in the solution of the scheduling problems we consider: the easiest phase, where one applies almost mechanically the Inclusion-Exclusion method by relaxing the problem and counting the relaxed schedules; and the most interesting phase, where the method is adapted to the specificities of the problem. About the specificities of the problems we consider, solving the generic $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem is a direct extension of solving the particular $P||C_{\max}$ problem, but solving the generic $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ problem requires introducing generating series and convolution.

3.1 Principles

In this chapter we consider a large class of parallel machine problems whose objectives are regular. As we have seen in section 1.1.5, semi-active schedules form a dominant set, and it is possible to represent them unambiguously by a list of jobs $(j_{ik})_{i,k}$ indexed by machines i and by chronological order k.

Before we solve the $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ and $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ problems, we have to tackle an issue: due to the presence of deadlines, it may happen that there is no solution. To determine whether or not there exists a solution, we remove the costs and replace them with the trivial, null costs, *i.e.* we take as objective function $\gamma = 0 = \max_j 0 = \sum_j 0$. So, we consider the fake optimization problem $\mathcal{P}^0_{opt} = R|r_{ij}, \tilde{d}_{ij}|$-, which has the same set of admissible schedules as the $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ and $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ problems. As we saw in section 2.1.1, an optimization problem reduces to a decision problem. To the fake optimization problem $\mathcal{P}^0_{opt}$ is associated a decision problem $\mathcal{P}^0_{dec}(\mathcal{I}, \varepsilon)$, which we will solve in section 3.2 or 3.3 p. 83. Then, $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ and $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ have a solution if and only if $\mathcal{P}^0_{opt}$ has a solution, if and only if $\mathcal{P}^0_{dec}(\mathcal{I}, 0)$ is true.

As noted in Remark 2.5 p. 49, the equations are simplified if one demonstrates that the problem is self-reducible. That is what we will do in section 3.1.1. Then, in section 3.1.2, we study as a fundamental example the $P||C_{\max}$ problem consisting in minimizing the makespan. Notice that it is equivalent to the Bin Packing problem, as we have seen in Example 1.6 p. 31. This fundamental example is very important for us, because it illustrates a difficulty: the direct relaxation of the coverage constraint leads to a suboptimal complexity. Karp [42] used a more subtle relaxation enabling to derive a much better complexity.

3.1.1 Self reducibility

We define the concatenation of two schedules as the concatenation of their job lists machine by machine. Thus, for a prefix π and a suffix σ representing schedules, we have:

$$\underbrace{\begin{pmatrix}(\pi_{1\,1},\ldots,\pi_{1\,h_1})\\ \vdots \\ (\pi_{m1},\ldots,\pi_{mh_m})\end{pmatrix}}_{\pi} \cdot \underbrace{\begin{pmatrix}(\sigma_{1\,1},\ldots,\sigma_{1\,\ell_1})\\ \vdots \\ (\sigma_{m1},\ldots,\sigma_{m\ell_m})\end{pmatrix}}_{\sigma} = \begin{pmatrix}(\pi_{1\,1},\ldots,\pi_{1\,h_1},\sigma_{1\,1},\ldots,\sigma_{1\,\ell_1})\\ \vdots \\ (\pi_{m1},\ldots,\pi_{mh_m},\sigma_{m1},\ldots,\sigma_{m\ell_m})\end{pmatrix} \tag{3.1}$$

Figure 3.1 shows how to make a parallel machine scheduling problem self-reducible: we consider, machine by machine, the last jobs of the prefix, which form what Jansen et al. [40] call the outline. The outline is therefore the vector $\begin{pmatrix}\pi_{1\,h_1}\\ \cdots \\ \pi_{mh_m}\end{pmatrix}$. We then define $\vec{B}$ as the vector formed by the completion times of the outline, which are also lower bounds on the starting times of all the jobs of the suffix.

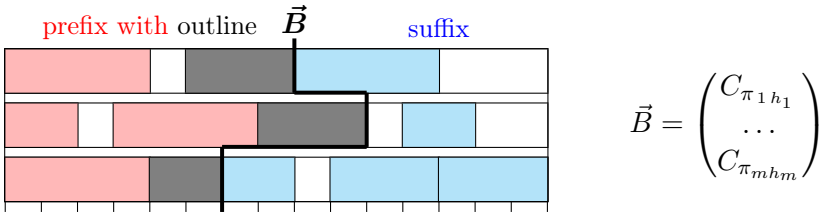


Figure 3.1: Self reducibility of a three parallel machine scheduling problem

From now on we will call $\vec{B}$ a "release bound front". A parallel machine scheduling problem is therefore self-reducible as soon as it takes into account release dates r_{ij} which depend on the job j but also on the machine i. Notice that taking $\vec{B}$ into account merely consists in replacing r_j with $\max(B_i, r_j)$.

The letter $\vec{B}$ has been chosen because it fits well with the vocabulary used in its definition: it is a Border or Barrier between the prefix and the suffix: each B_i is a lower Bound on the release dates of all suffix operations on machine i and machines are Busy Before $\vec{B}$.

We need an upper bound, say $B_{\max}$, for the components of all possible release bound fronts. Since a release bound front is also a completion time front, we just have to take an upper bound from the makespan of any schedule (*i.e.* $\forall\, S : \text{schedule}, C_{\max}(S) \leqslant B_{\max}$). So, we define $B_{\max}$ by this formula, corresponding to the worst case where the first job is the one with the longest release time:

$$B_{\max} = \max_i \Big(\max_j r_{ij} + \sum_j p_{ij} \Big) \tag{3.2}$$

Notice that $B_{\max} = O^*(||\mathcal{I}||)$. This will be useful for bounding the number of states of dynamic programming schemes.

To solve the optimization problem, *i.e.* to compute an explicit optimal solution, we propose a straightforward adaptation of Algorithm 2.3 p. 48. We denote by $(i : j)$ the schedule containing a single job j on machine i, *i.e.* the list of job lists containing, at position i the list (j) with the single job j, and empty lists elsewhere. We also denote by $()$ the empty prefix, actually the list of empty job lists. Using these notations, Algorithm 3.2 shows the adapted algorithm, using function *OptimumObjective* implemented by Algorithm 2.2 p. 48 to compute the optimal objective value γ^{opt}:

Algorithm 3.2: Computation of an optimal schedule on parallel machines

Function *OptimalSolution*($\mathcal{I}$)**:**
  $\gamma^{opt} \leftarrow$ *OptimumObjective*($\mathcal{I}$)
  $\pi \leftarrow ()$
  repeat n *times*
    find $i \in \{1, \ldots, m\}$, $j \notin \pi$ **such that** $\mathcal{P}_{dec}(\mathcal{I}, \pi \cdot (i : j), \gamma^{opt})$
    $\pi \leftarrow \pi \cdot (i : j)$
  return π

3.1.2 Makespan minimization on identical machines

In this section we focus on the $P||C_{\max}$ problem, equivalent to the Bin Packing problem as seen in Example 1.6 p. 31. Machines are identical, there are neither release times nor deadlines, and the objective to minimize is the makespan.

As shown in section 2.1.1, in order to solve the $\mathcal{P}_{opt} = P||C_{\max}$ problem, it is sufficient to solve, for each schedule prefix π and threshold makespan ε, the decision problem $\mathcal{P}_{dec}(\mathcal{I}, \pi, \varepsilon)$ defined by Equation (2.1) p. 47. To implement this decision problem, we count the number $N(\vec{B}, J', \varepsilon)$ of admissible schedules formed of jobs of J', starting from release bound front $\vec{B}$ and with makespan at most ε. As shown in Figure 3.1 p. 76, imposing a prefix amounts to imposing that the remaining jobs are outside the prefix and have release time bounds B_i equal to the completion times of the prefix on each machine. We derive:

$$\mathcal{P}_{dec}\left(\mathcal{I},\ \begin{pmatrix}(\pi_{1\,1}, \ldots, \pi_{1\,h_1})\\ \cdots \\ (\pi_{m1}, \ldots, \pi_{mh_m})\end{pmatrix},\ \varepsilon\right) \iff N\left(\begin{pmatrix}C_{\pi_{1\,h_1}}\\ \cdots \\ C_{\pi_{mh_m}}\end{pmatrix},\ J \setminus \pi,\ \varepsilon\right) > 0 \tag{3.3}$$

We now compare two ways of applying the Inclusion-Exclusion formula to compute $N(\vec{B}, J', \varepsilon)$.

Inclusion-Exclusion: a first attempt

The (strict) schedules we consider are two-dimensional job lists covering the job set J' and with makespan at most ε. We relax the coverage constraint. The set of admissible relaxed schedules $\mathscr{R}$ is composed of job lists with makespan at most ε, containing potentially duplicate or missing jobs, but with exactly $n' = |J'|$ jobs. As we consider two dimensional

lists, this means that the sum of the numbers of jobs ℓ_i per machine is equal to n'. We derive:

$$\mathscr{R} = \left\{ (j_{ik})_{1 \leqslant i \leqslant m, 1 \leqslant k \leqslant \ell_i} \;\middle|\; \sum_{i=1}^{m} \ell_i = n' \quad \wedge \quad \forall\, i, C_{j_i \ell_i} \leqslant \varepsilon \right\} \tag{3.4}$$

From a scheduling point of view, this means that our admissible relaxed schedules are usual parallel machine schedules, where jobs are distributed over the machines, with the special property that jobs may be absent or duplicated. Figure 3.3 shows an example with n'=6 jobs scheduled after the release bound front $\vec{B}$:

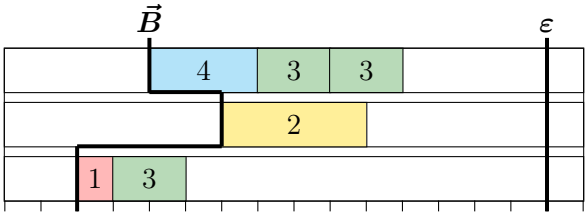


Figure 3.3: A sample relaxed schedule with exactly 6 jobs

For a job subset $X \subset J'$ and a threshold makespan ε, we define $N_{X,\varepsilon}[\vec{B}, \ell]$ as the number of admissible relaxed schedules with exactly ℓ jobs, using only jobs from X, starting from $\vec{B}$, and with makespan at most ε. Then, we derive from Formula (2.17) p. 53:

$$N(\vec{B}, J', \varepsilon) = \sum_{X \subset J'} (-1)^{|J'|-|X|} N_{X,\varepsilon}[\vec{B}, n'] \tag{3.5}$$

We now just have to compute $N_{X,\varepsilon}[\vec{B}, \ell]$ by dynamic programming. For $\ell = 0$ there is only one schedule, the empty one, and it is admissible. For $\ell > 0$, we try each possible job j in the first position on each possible machine i. There are admissible schedules to count if the completion time C_j of job j is not larger than ε, in which case it becomes the release bound of the suffix jobs scheduled on machine i, and we recursively count the possible suffixes. We derive:

$$N_{X,\varepsilon}[\vec{B}, 0] = 1 \tag{3.6}$$

$$N_{X,\varepsilon}[\vec{B}, \ell] = \sum_{i=1}^{m} \sum_{\substack{j \in X \\ C_j \leqslant \varepsilon \\ \text{where } C_j = B_i + p_j}} N_{X,\varepsilon}[\vec{B}', \ell - 1] \quad \text{where} \quad \left\{ \begin{array}{l} B'_{i'} = B_{i'} \text{ for } i' \neq i \\ B'_i = C_j \end{array} \right\} \quad \text{for } \ell \neq 0 \tag{3.7}$$

Because each B_i is bounded by $||\mathcal{I}||$, the number of states of this dynamic programming scheme, and therefore the time and space complexity bounds to compute each separate $N_{X,\varepsilon}$, are in $O^*(||\mathcal{I}||^m)$. We are about to describe a much more efficient relaxation and dynamic programming scheme.

Inclusion-Exclusion: a better solution

The method we now describe is due to Karp [42]. The presentation we adopt is very different from his article, for two reasons: on the one hand, Karp focuses on the Bin Packing problem whereas we focus on the equivalent scheduling problem. On the other hand, Karp uses a backward dynamic programming scheme to count relaxed solutions, whereas, as stated at the end of section 2.1.1, we prefer forward schemes.

As in the first attempt, the strict schedules we consider are two-dimensional job lists covering the job set J' and with makespan at most ε. Again, we take $n' = |J'|$ and we relax the coverage constraint, but in addition we apply a further relaxation: instead of requiring to schedule exactly n' jobs distributed on all machines, we require to schedule at most n' jobs per machine.

From a mathematical point of view, this means that we consider a very simple list shape, as defined in section 2.1.3: our schedules are elements of J'^* where J'^* is the set of lists of m lists of at most n' jobs, *i.e.* $J'^* = \left(J'^{(n')}\right)^m$.

In comparison, we used a much more complicated set of list shapes in the first attempt: the corresponding J'^* was implicitly the union of products of job lists whose sum of lengths equals n', *i.e.* $J'^* = \bigcup_{\ell_1+\ldots+\ell_m=n'} J'^{\ell_1} \times \cdots \times J'^{\ell_m}$.

The set of admissible relaxed schedules $\mathscr{R}$ is composed of job lists with makespan at most ε, containing potentially duplicate or missing jobs, with at most n' jobs per machine. We derive:

$$\mathscr{R} = \left\{ (j_{ik})_{1\leqslant i\leqslant m, 1\leqslant k\leqslant \ell_i} \;\middle|\; \forall i, \ell_i \leqslant n' \quad \wedge \quad \forall i, C_{j_{i\,\ell_i}} \leqslant \varepsilon \right\} \tag{3.8}$$

From a scheduling point of view, this means that our admissible relaxed schedules may have from 0 to $m \cdot n'$ jobs, potentially absent or duplicated. Figure 3.4 shows an example with n'=6 jobs scheduled after the release bound front $\vec{B}$, to be compared with Figure 3.3 p. 78.

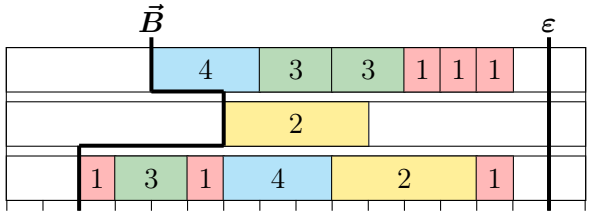


Figure 3.4: A sample relaxed schedule with at most 6 jobs per machine

We define the number $N'(\vec{B}, J', \varepsilon)$ of covering relaxed admissible schedules formed of jobs of J', starting from $\vec{B}$ and with makespan at most ε. We must be very careful because, due to the relaxation of list lengths, covering relaxed admissible schedules are not exactly strict admissible schedules *i.e.* $N'(\vec{B}, J', \varepsilon) \neq N(\vec{B}, J', \varepsilon)$.

Any strict schedule is also a covering relaxed admissible schedule, so $N'(\vec{B}, J', \varepsilon) \geqslant N(\vec{B}, J', \varepsilon)$. On the other way, given a covering relaxed admissible schedule, we can derive a strict admissible schedule with a lower or equal makespan by removing any of its duplicate jobs. So, $N'(\vec{B}, J', \varepsilon) > 0 \implies N(\vec{B}, J', \varepsilon) > 0$. Finally, $N'(\vec{B}, J', \varepsilon) > 0 \iff N(\vec{B}, J', \varepsilon) > 0$. Following Definition 2.1 p. 47, we can safely compute $N'(\vec{B}, J', \varepsilon)$ instead of $N(\vec{B}, J', \varepsilon)$, and test whether $N'(\vec{B}, J', \varepsilon) > 0$ to implement the $\mathcal{P}_{dec}$ decision problem.

For a job subset $X \subset J'$ and a threshold makespan ε, we define $N'_{X,\varepsilon}(\vec{B})$ as the number of admissible relaxed schedules using only jobs from X, starting from $\vec{B}$, and with makespan at most ε. Then, we derive from Formula (2.17) p. 53:

$$N'(\vec{B}, J', \varepsilon) = \sum_{X \subset J'} (-1)^{|J'|-|X|} N'_{X,\varepsilon}(\vec{B}) \tag{3.9}$$

The special relaxation of list lengths removes any correlation between job lists of different machines. It is a way of transforming a m-machine parallel problem into m independent single-machine problems. We define $M'_{X,\varepsilon}[b, \ell]$ as the number of admissible relaxed schedules on X, starting at release time bound b on a single machine, with at most ℓ jobs, and with makespan at most ε. The maximum objective is globally bounded by ε if and only if it is bounded by ε on each machine, hence we have:

$$N'_{X,\varepsilon}(\vec{B}) = \prod_{i=1}^{m} M'_{X,\varepsilon}[B_i, n'] \tag{3.10}$$

We now just have to compute $M'_{X,\varepsilon}[b, \ell]$ by dynamic programming. For $\ell = 0$ there is only one schedule, the empty one, and it is admissible. For $\ell > 0$, we try each possible job j in the first position. There are admissible schedules to count if the completion time C_j of job j is not larger than ε, in which case it becomes the release bound of the other jobs, and we recursively count the possible suffixes. We derive:

$$M'_{X,\varepsilon}[b, 0] = 1 \tag{3.11}$$

$$M'_{X,\varepsilon}[b, \ell] = \sum_{\substack{j \in X \\ C_j \leqslant \varepsilon \\ \text{where } C_j = B_i + p_j}} M'_{X,\varepsilon}[C_j, \ell - 1] \qquad \text{for } \ell \neq 0 \tag{3.12}$$

Because b is bounded by $||\mathcal{I}||$, the number of states of this dynamic programming scheme, and therefore the time and space complexity bounds to compute each separate $M'_{X,\varepsilon}$ and their product $N'_{X,\varepsilon}$ are in $O^*(||\mathcal{I}||)$. This is far better than the first attempt, and we shall generalize this technique to other parallel machine schedules.

3.2 Minimizing a maximum cost

In this section, we focus on the $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem. We solve this problem by using the same technique as for the $P||C_{\max}$ problem tackled in section 3.1.2.

3.2.1 Problem relaxation

In order to solve the $\mathcal{P}_{opt} = R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem, it is sufficient to solve, for each prefix π and objective threshold ε, the decision problem $\mathcal{P}_{dec}(\mathcal{I}, \pi, \varepsilon)$ defined by Equation (2.1) p. 47. To implement $\mathcal{P}_{dec}$, We define the number $N(\vec{B}, J', \varepsilon)$ of (strict) admissible schedules formed of jobs of J', starting from $\vec{B}$ and with objective at most ε. As shown in Figure 3.1 p. 76, imposing a prefix amounts to imposing that the remaining jobs are outside the prefix and have release time bounds B_i equal to the completion times of the prefix on each machine.

$$\mathcal{P}_{dec}\left(\mathcal{I},\ \begin{pmatrix}(\pi_{1\,1},\ldots,\pi_{1\,h_1})\\ \cdots\\ (\pi_{m1},\ldots,\pi_{mh_m})\end{pmatrix},\ \varepsilon\right) \iff N\left(\begin{pmatrix}C_{\pi_{1\,h_1}}\\ \cdots\\ C_{\pi_{mh_m}}\end{pmatrix},\ J\setminus\pi,\ \varepsilon\right) > 0 \tag{3.13}$$

The strict schedules we consider are two-dimensional job lists covering the job set J' and with objective at most ε. We set $n' = |J'|$, relax the coverage constraint, and allow to schedule at most n' jobs per machine (instead of exactly n' jobs globally). So, the job lists we consider are elements of $J'^* = \left(J'^{(n')}\right)^m$. The set of admissible relaxed schedules $\mathscr{R} \subset J'^*$ is composed of job lists containing potentially duplicate or missing jobs, with at most n' jobs per machine, with no deadline violation, and with objective *i.e.* maximum cost $\gamma = \max f_{ij}$ at most ε. We denote by ℓ_i the actual number of jobs on machine i and we derive:

$$\mathscr{R} = \left\{(j_{ik})_{1\leqslant i\leqslant m, 1\leqslant k\leqslant \ell_i} \;\middle|\; \forall\, i, \ell_i \leqslant n' \ \wedge\ \forall\, i,k, C_{j_{ik}} \leqslant \tilde{d}_{i,j_{ik}} \ \wedge\ \forall\, i,k, f_{ij_{ik}}(C_{j_{ik}}) \leqslant \varepsilon\right\} \tag{3.14}$$

We define the number $N'(\vec{B}, J', \varepsilon)$ of covering relaxed admissible schedules formed of jobs of J', starting from $\vec{B}$ and with objective at most ε. Due to the relaxation of list lengths, covering relaxed admissible schedules are not exactly strict relaxed schedules, but a strict schedule is also a covering relaxed admissible schedule, and, given a covering relaxed admissible schedule, we can derive a strict relaxed schedule with a lower or equal makespan value by removing any of its duplicate jobs. So, the number N' is suitable to solve the $\mathcal{P}_{dec}$ decision problem as defined by Equation (3.13).

3.2.2 Counting Relaxed Schedules

For a job subset $X \subset J'$ and a threshold makespan ε, we define $N'_{X,\varepsilon}(\vec{B})$ as the number of admissible relaxed schedules using only jobs from X, starting from $\vec{B}$, and with objective value at most ε. Then, we derive from Formula (2.17) p. 53:

$$N'(\vec{B}, J', \varepsilon) = \sum_{X\subset J'} (-1)^{|J'|-|X|} N'_{X,\,\varepsilon}(\vec{B}) \tag{3.15}$$

We define $M'_{X,i,\varepsilon}[b,\ell]$ as the number of admissible relaxed schedules on X, starting at release time bound b on the single machine i, with at most ℓ jobs, and with objective value at most ε. The maximum objective value is globally bounded by ε if and only if it is

bounded by ε on each machine, hence we have:

$$N'_{X,\varepsilon}(\vec{B}) = \prod_{i=1}^{m} M'_{X,i,\varepsilon}[B_i, n'] \tag{3.16}$$

We now just have to compute $M'_{X,i,\varepsilon}[b,\ell]$ by dynamic programming. For $\ell = 0$ there is only one schedule, the empty one, and it is admissible. For $\ell > 0$, we try each possible job j in the first position. There are admissible schedules to count if j does not violate its deadline and if the cost of job j is not larger than ε, in which case the completion time of j becomes the release bound of the other jobs, and we recursively count the possible suffixes. We derive:

$$M'_{X,i,\varepsilon}[b,0] = 1 \tag{3.17}$$

$$M'_{X,i,\varepsilon}[b,\ell] = \sum_{\substack{j\in X \\ C_j \leqslant \tilde{d}_{ij} \\ f_{ij}(C_j)\leqslant\varepsilon \\ \text{where } C_j=\max(B_i,r_{ij})+p_j}} M'_{X,i,\varepsilon}[C_j, \ell-1] \qquad \text{for } \ell \neq 0 \tag{3.18}$$

3.2.3 Worst-case complexity bounds

Our algorithms are now completely described, and we are about to evaluate their worst-case complexities. We must be careful, because they manipulate big integers, and we cannot assume constant time and space arithmetic operations. An arithmetic operation has a quasi-linear time and space complexity with respect to the logarithm (or size in bits) of the result.

Lemma 3.1. The size in bits of each $M_{X,i,\varepsilon}[b,\ell]$ is polynomially bounded: $\log M_{X,i,\varepsilon}[b,\ell] = O^*(n \log n) = O^*(1)$.

Proof. There are at most n jobs in a sequence and n values per job, so $M_{X,i,\varepsilon}[b,\ell] = O^*(n^n)$, thus $\log M_{X,i,\varepsilon}[b,\ell] = O^*(\log n^n) = O^*(n \log n)$. □

Lemma 3.2. Each $M_{X,i,\varepsilon}[B_i, n']$ involved in Formula (3.16) can be computed in $O^*(||\mathcal{I}||)$ time and space.

Proof. Each value of b involved in the dynamic programming scheme to compute $M_{X,i,\varepsilon}[B_i, n']$ is bound by $B_{\max}$ as defined by Equation (3.2) p. 76, so dynamic programming requires computing of $O^*(B_{\max} \times n')$ values, each of size $O^*(1)$, hence a time and space complexity in $O^*(B_{\max} \times n' \times 1) = O^*(B_{\max}) = O^*(||\mathcal{I}||)$. □

Remark 3.3. Following equations of Karp [42], removing the length parameter ℓ in the dynamic programming states, *i.e.* computing $M_{X,i,\varepsilon}[b]$ instead of $M_{X,i,\varepsilon}[b,\ell]$ may appear as an obvious simplification. But it makes complexity worse by allowing sequences of any length, instead of allowing sequences of length at most $n' \leqslant n$. Consider a fixed machine i and suppose there are two jobs $j \neq j'$ with $p_{ij} = p_{ij'} = 1$ and $f_{ij}(C) \leqslant \varepsilon, f_{ij'}(C) \leqslant \varepsilon$

for any C. Starting from $b = 0$, any sequence $(j_1 \in \{j, j'\}, \ldots, j_{B_{\max}} \in \{j, j'\})$ is valid, hence $M_{X,i,\varepsilon}[0] \geqslant 2^{B_{\max}}$, so $\log M_{X,i,\varepsilon}[0] \geqslant B_{\max} = \Theta(||\mathcal{I}||)$, and all complexities have to be multiplied by the instance measure $||\mathcal{I}||$.

Proposition 3.4. An optimal schedule can be computed in $O^*(2^n||\mathcal{I}||)$ time and $O^*(||\mathcal{I}||)$ space.

Proof. To proceed, we use Algorithm 2.2 p. 48 to compute γ^{opt} and Algorithm 3.2 p. 77 to compute an optimal schedule, which imply $O^*(\log \gamma^{opt}) = O^*(\log ||\mathcal{I}||) = O^*(1)$ computations of $N'(\vec{B}, J', \varepsilon)$. Each computation of $N'(\vec{B}, J', \varepsilon)$ is a sum over the $2^{n'}$ subsets of J and requires $2^{n'} \leqslant 2^n$ computations of $N_X(\vec{0}, \varepsilon)$. Each $N_X(\vec{0}, \varepsilon)$ requires (products and) computation of the $m = O^*(1)$ factors $M_{X,i,\varepsilon}[B_i, n']$, each in $O^*(||\mathcal{I}||)$ time. Hence the time complexity in $O^*(2^n) \times O^*(||\mathcal{I}||) \times O^*(1) = O^*(2^n||\mathcal{I}||)$. The space complexity is dominated by the computation of $M_{X,i,\varepsilon}[B_i, n']$, in $O^*(||\mathcal{I}||)$. □

3.3 Minimizing a total cost

In this section, we focus on the $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ problem. While our solution of this problem relies on Inclusion-Exclusion, counting relaxed schedules to minimize a total cost is radically different from counting relaxed schedules to minimize a maximum cost.

3.3.1 Problem relaxation

In order to solve the $\mathcal{P}_{opt} = R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ problem, it is sufficient to solve, for each prefix π and objective threshold ε, the decision problem $\mathcal{P}_{dec}(\mathcal{I}, \pi, \varepsilon)$ defined by Equation (2.1) p. 47. To implement $\mathcal{P}_{dec}$, We define the number $N(\vec{B}, J', \varepsilon)$ of (strict) admissible schedules formed of jobs of J', starting from $\vec{B}$ and with objective at most ε. As shown in Figure 3.1 p. 76, imposing a prefix amounts to imposing that the remaining jobs are outside the prefix and have release time bounds B_i equal to the completion times of the prefix on each machine. Notice that, because the objective is a sum of elementary costs, the jobs of the prefix consume a part of the total available objective ε and this part has to be subtracted from ε.

$$\mathcal{P}_{dec}\left(\mathcal{I},\ \begin{pmatrix}(\pi_{1\,1},\ldots,\pi_{1\,h_1})\\ \cdots \\ (\pi_{m1},\ldots,\pi_{mh_m})\end{pmatrix},\ \varepsilon\right) \iff N\left(\begin{pmatrix}C_{\pi_{1\,h_1}}\\ \cdots \\ C_{\pi_{mh_m}}\end{pmatrix},\ J \setminus \pi,\ \varepsilon - \sum_{\substack{1\leqslant i\leqslant m\\ 1\leqslant k\leqslant h_i}} f_{i\pi_{ik}}(C_{\pi_{ik}})\right) > 0 \quad (3.19)$$

The strict schedules we consider are two-dimensional job lists covering the job set J' and with objective at most ε. We set $n' = |J'|$, relax the coverage constraint, and allow to schedule at most n' jobs per machine (instead of exactly n' jobs globally). So, the job lists we consider are elements of $J'^* = \left(J'^{(n')}\right)^m$. The set of admissible relaxed schedules $\mathscr{R} \subset J'^*$ is composed of job lists containing potentially duplicate or missing jobs, with at most n' jobs per machine, with no deadline violation, and with objective *i.e.* total cost

$\gamma = \sum f_{ij}$ at most ε. We denote by ℓ_i the actual number of jobs on machine i and we derive:

$$\mathscr{R} = \left\{ (j_{ik})_{1 \leqslant i \leqslant m, 1 \leqslant k \leqslant \ell_i} \;\middle|\; \forall i, \ell_i \leqslant n' \;\wedge\; \forall i,k, C_{j_{ik}} \leqslant \tilde{d}_{i,j_{ik}} \;\wedge\; \sum_{i,k} f_{ij_{ik}}(C_{j_{ik}}) \leqslant \varepsilon \right\} \tag{3.20}$$

We define the number $N'(\vec{B}, J', \varepsilon)$ of covering relaxed admissible schedules formed of jobs of J', starting from $\vec{B}$ and with objective at most ε. Although covering relaxed admissible schedules are not exactly strict relaxed schedules, the number N' is suitable to solve the $\mathcal{P}_{dec}$ decision problem as defined by Equation (3.19) p. 83.

Unfortunately, the number $N'(\vec{B}, J', \varepsilon)$ is too difficult to compute so, instead of computing it, we compute the number $N^{=}(\vec{B}, J', \varepsilon)$ of covering relaxed admissible schedules formed of jobs of J', starting from $\vec{B}$ and with objective exactly equal to ε. We must be very careful, because $N^{=}$ lacks some nice properties of N': a complete relaxed solution corresponds to a strict solution with a potentially smaller, not necessarily equal objective value; the condition $(N^{=}(\vec{B}, J', \varepsilon) > 0)$ is not equivalent to $(N(\vec{B}, J', \varepsilon) > 0)$, so $N^{=}$ is not an implementation of $\mathcal{P}_{dec}$. Notice however that N' and $N^{=}$ are related by simple formulas:

$$N^{=}(\vec{B}, J', \varepsilon) = N'(\vec{B}, J', \varepsilon) - N'(\vec{B}, J', \varepsilon - 1) \tag{3.21}$$

$$N'(\vec{B}, J', \varepsilon) = \sum_{\varepsilon' \leqslant \varepsilon} N^{=}(\vec{B}, J', \varepsilon') \tag{3.22}$$

These formulas are useful from a theoretical point of view, but from a computational point of view, using them as-is to derive N' from $N^{=}$ would uselessly increase the time complexity bound, because of the summation for all $\varepsilon' \leqslant \varepsilon$. So, it is better to derive an optimal schedule directly from $N^{=}$ without computing N' as an intermediate value. We hereafter describe how to compute $N^{=}(\vec{B}, J', \varepsilon)$ and how to adapt the rationale to derive an optimal schedule.

3.3.2 Counting Relaxed Schedules

For a job subset $X \subset J'$ and a threshold makespan ε, we define $N^{=}_{X}(\vec{B}, \varepsilon)$ as the number of admissible relaxed schedules using only jobs from X, starting from $\vec{B}$, and with objective value exactly ε. Then, we derive from Formula (2.17) p. 53:

$$N^{=}(\vec{B}, J', \varepsilon) = \sum_{X \subset J'} (-1)^{|J'|-|X|} N^{=}_{X}(\vec{B}, \varepsilon) \tag{3.23}$$

As we will see later, instead of applying this formula separately to each $N^{=}_{X}(\vec{B}, \varepsilon)$, we will group several $N^{=}_{X}(\vec{B}, \varepsilon)$ together to form a vector and apply a vectorial version of this formula.

The algorithm to compute $N^{=}_{X,\varepsilon}(\vec{B})$ relies on the following relation: the total objective value of a schedule is equal to ε if and only if there exists $\varepsilon_1 \dots \varepsilon_m$ such that $\varepsilon_1 + \dots + \varepsilon_m = \varepsilon$ and the total objective on each machine i is equal to ε_i. Formally:

$$\sum_{i,j} f_{ij} = \varepsilon \iff \sum_i \underbrace{\left(\sum_j f_{ij}\right)}_{\varepsilon_i} = \varepsilon$$
$$\iff \exists\, \varepsilon_1 \dots \varepsilon_m \;\Big|\; \sum_i \varepsilon_i = \varepsilon \quad \wedge \quad \forall\, i, \sum_j f_{ij} = \varepsilon_i \tag{3.24}$$

We define $M_{X,i}[b, \ell, \varepsilon]$ as the number of admissible relaxed schedules on X, starting at release time bound b on the single machine i, with at most ℓ jobs, and with objective value exactly ε. We set $n' = |J'|$, and we derive from relation (3.24):

$$N^{=}_{X}(\vec{B}, \varepsilon) = \sum_{\varepsilon_1 + \dots + \varepsilon_m = \varepsilon} M_{X,1}[B_1, n', \varepsilon_1] \times \dots \times M_{X,m}[B_m, n', \varepsilon_m] \tag{3.25}$$

This is a convolution, or Cauchy product, corresponding to the product of generating power series on an indeterminate Z, where each $N^{=}_{X}(\vec{B}, \varepsilon)$ and each $M_{X,i}[B_i, n', \varepsilon]$ is the coefficient of Z^ε, which leads to:

$$\left(\sum_{\varepsilon \in \mathbb{N}} N^{=}_{X}(\vec{B}, \varepsilon) Z^\varepsilon\right) = \prod_{i=1}^{m} \left(\sum_{\varepsilon \in \mathbb{N}} M_{X,i}[B_i, n', \varepsilon] Z^\varepsilon\right) \tag{3.26}$$

From a computational point of view, it is convenient to fix a maximum objective $\varepsilon_{\max}$, and to restrict to $\varepsilon \leqslant \varepsilon_{\max}$, which corresponds to multiplying polynomials modulo $Z^{\varepsilon_{\max}+1}$. We identify a vector $(U_0, \dots, U_{\varepsilon_{\max}})$ and the polynomial $U = \sum_{\varepsilon=0}^{\varepsilon_{\max}} U_\varepsilon Z^\varepsilon$.

The standard convolution operator $*$ takes two polynomials U and V, and computes the polynomial W such that $W = U * V = UV \mod Z^{\varepsilon_{\max}+1}$. So, conceptually, a convolution is a mere polynomial product (with modulo), but practically a convolution explicitly calculates the coefficients of this polynomial product:

$$\left(\overbrace{\sum_{\varepsilon=0}^{\varepsilon_{\max}} U_\varepsilon Z^\varepsilon}^{U}\right)\left(\overbrace{\sum_{\varepsilon=0}^{\varepsilon_{\max}} V_\varepsilon Z^\varepsilon}^{V}\right) \mod Z^{\varepsilon_{\max}+1} = \overbrace{\sum_{\varepsilon=0}^{\varepsilon_{\max}} \underbrace{\left(\sum_{\alpha+\beta=\varepsilon} U_\alpha V_\beta\right)}_{(U*V)_\varepsilon} Z^\varepsilon}^{U*V} \tag{3.27}$$

That is: $\forall\, \varepsilon \leqslant \varepsilon_{\max}, (U * V)_\varepsilon = \sum_{\alpha+\beta=\varepsilon} U_\alpha V_\beta$. Notice how this convolution formula resembles Formula (3.25), justifying our interest for generating series and convolution.

Computing $U * V$ by straightforward evaluation of each $(U * V)_\varepsilon$ requires $O(\varepsilon_{max}^2)$ operations, but $U * V$ can be computed in $O(\varepsilon_{max} \log \varepsilon_{max} \log\log \varepsilon_{max})$ by using the traditional algorithm of Schönhage and Strassen [77] and even in $O(\varepsilon_{max} \log \varepsilon_{max})$ by using the very recent algorithm of Harvey and Van Der Hoeven [34].

To apply the convolution formula we need vectors indexed by ε. Let us define $\vec{N}^{=}_{X}(\vec{B})=\big(N^{=}_{X}(\vec{B},0),\ldots,N^{=}_{X}(\vec{B},\varepsilon_{\max})\big)$ and $\vec{M}_{X,i}(b)=\big(M_{X,i}[b,n',0],\ldots,M_{X,i}[b,n',\varepsilon_{\max}]\big)$. Using the standard convolution operator $*$, we have:

$$\vec{N}^{=}_{X}(\vec{B}) = \vec{M}_{X,1}(B_1) * \ldots * \vec{M}_{X,m}(B_m) \tag{3.28}$$

Due to this convolution formula, computing $\vec{N}^{=}_{X}(\vec{B})$, *i.e.* all $N^{=}_{X}(\vec{B},\varepsilon)$ together for $\varepsilon \leqslant \varepsilon_{\max}$, is not more costly than computing $N_X(\vec{B},\varepsilon_{\max})$ alone. We must apply the Inclusion-Exclusion formula as a sum of vectors, instead of applying the scalar formula for each ε. Our aim is to compute $N^{=}(\vec{B},J',\varepsilon)$ for any ε, so we define the vector $\vec{N}^{=}(\vec{B},J') = \big(N^{=}(\vec{B},J',0),\ldots,N^{=}(\vec{B},J',\varepsilon_{\max})\big)$ and we derive from Formula (2.17) p. 53:

$$\vec{N}^{=}(\vec{B},J') = \sum_{X\subset J'} (-1)^{|J'|-|X|}\vec{N}^{=}_{X}(\vec{B}) \tag{3.29}$$

To finalize the computation of each $N^{=}(\vec{B},J',\varepsilon)$, we now just have to compute $M_{X,i}[b,\ell,\varepsilon]$ by dynamic programming. Notice that, for a fixed X and a fixed i, all $M_{X,i}[b,\ell,\varepsilon]$ together *i.e.* the whole vector $\vec{M}_{X,i}(b)$ involved in Formula (3.28) can be computed at once inside the same dynamic programming scheme.

For $\ell = 0$ there is only one schedule, the empty one, and its objective is null, so it counts as one relaxed admissible schedule if $\varepsilon = 0$ and no relaxed admissible schedule otherwise, which we write $\mathbb{1}_{\varepsilon=0}$. For $\ell > 0$, we try each possible job j in the first position. There are admissible schedules to count if j does not violate its deadline and if the cost of job j is not larger than ε. In this case, this cost has to be subtracted from ε and the completion time of j becomes the release bound of the other jobs, and we recursively count the possible suffixes. We derive:

$$M_{X,i}[b,0,\varepsilon] = \mathbb{1}_{\varepsilon=0} \tag{3.30}$$

$$M_{X,i}[b,\ell,\varepsilon] = \sum_{\substack{j\in X\\ C_j\leqslant \tilde{d}_{ij}\\ f_{ij}(C_j)\leqslant\varepsilon\\ \text{where } C_j=\max(B_i,r_{ij})+p_j}} M'_{X,i}[C_j,\ell-1,\varepsilon-f_{ij}(C_j)] \qquad \text{for } \ell\neq 0 \tag{3.31}$$

3.3.3 Optimal schedule computation

The whole procedure we just described to compute $N^{=}(\vec{B},J',\varepsilon)$ for any ε relies on $\varepsilon_{\max}$ being an upper bound on all the possible values of ε we need. Moreover, Algorithm 2.2 p. 48 is powerless to compute the optimum objective value γ^{opt} since it relies on N' instead of $N^{=}$. So, the rationale to determine the optimum objective value has to be deeply revised.

We can actually compute both $\varepsilon_{\max}$ and γ^{opt} as follows: choose an arbitrary $\varepsilon_{\max}$ and compute $\vec{N}^{=}(\vec{B},J')$ for this $\varepsilon_{\max}$, *i.e.* $\vec{N}^{=}(\vec{B},J') = \big(N^{=}(\vec{B},J',0),\ldots,N^{=}(\vec{B},J',\varepsilon_{\max})\big)$. Then, the minimum objective is the lowest ε, if it exists, such that $N^{=}(\vec{B},J',\varepsilon)$ is not null. If no such ε exists, this means that $\varepsilon_{\max}$ is too small and must be increased. During this computation, we don't need any prefix π, because such a prefix is only required to

compute an optimal schedule. So, we only need to take $\vec{B} = \vec{0}$, *i.e.* no release bound, and $J' = J = \{1, \ldots, n\}$, *i.e.* the whole set of jobs. The computation of an optimal schedule is given in Algorithm 3.5.

Algorithm 3.5: Computation of the optimum objective value for a total cost objective

Function *OptimumSumObjective*:
  $\vec{B} \leftarrow \vec{0}$
  $J' \leftarrow \{1, \ldots, n\}$
  $\varepsilon_{\max} \leftarrow 1$
  $\vec{N}^{=} \leftarrow (N^{=}(\vec{B}, J', 0), N^{=}(\vec{B}, J', \varepsilon_{\max}))$
  while $\vec{N}^{=} = \vec{0}$ **do**
    $\varepsilon_{\max} \leftarrow 2\ \varepsilon_{\max}$
    $\vec{N}^{=} \leftarrow (N^{=}(\vec{B}, J', 0), \ldots, N^{=}(\vec{B}, J', \varepsilon_{\max}))$
  $\gamma^{opt} \leftarrow \min\{0 \leqslant \varepsilon \leqslant \varepsilon_{\max} \mid (\vec{N}^{=})_{\varepsilon} > 0\}$
  return γ^{opt}

Once γ^{opt} has been computed, we can safely take $\varepsilon_{\max} = \gamma^{opt}$ for further computations. To compute an optimal solution, we call Algorithm 3.2 p. 77 (itself a straightforward adaptation of Algorithm 2.3 p. 48), which calls $\mathcal{P}_{dec}$, which is in turn calls N'. We derive $N'(\vec{B}, J', \varepsilon)$ from $\vec{N}^{=}(\vec{B}, J', \varepsilon)$ using Formula (3.22) p. 84. Notice that this formula is in fact trivial and does not correspond to an actual computation of N', because, in the context of Algorithm 3.2 p. 77, ε is the smallest possible objective value for covering schedules starting from $\vec{B}$ and with jobs in J. So, we have: $\forall \varepsilon' < \varepsilon, N^{=}(\vec{B}, J', \varepsilon) = 0$ and thus $N'(\vec{B}, J', \varepsilon) = N^{=}(\vec{B}, J', \varepsilon)$.

3.3.4 Worst-case complexity bounds

Lemma 3.5. For a given $\varepsilon_{\max}$, all $M_{X,i}[b, \ell, \varepsilon]$ for $\varepsilon \leqslant \varepsilon_{max}$ can be computed together in $O^*(||\mathcal{I}||\varepsilon_{\max})$ time and space.

Proof. The dynamic programming scheme common to all $M_{X,i}$ requires to compute $O^*(B_{\max} \times n \times \varepsilon_{\max})$ values, each value being a big integer of size $O^*(1)$, hence a time and space complexity in $O^*(B_{\max} \times n \times \varepsilon_{\max} \times 1) = O^*(B_{\max} \times \varepsilon_{\max}) = O^*(||\mathcal{I}|| \times \varepsilon_{\max})$. □

Lemma 3.6. For a given $\varepsilon_{\max}$, each vector $\vec{N}^{=}_{X}(\vec{B})$ indexed by $\varepsilon \leqslant \varepsilon_{max}$ can be computed in $O^*(||\mathcal{I}||\varepsilon_{\max})$ time and space.

Proof. Computing $\vec{N}^{=}_{X}(\vec{B})$ requires computation of all $M_{X,i}[b, \ell, \varepsilon]$, and $O(m) = O^*(1)$ convolutions on vectors of length $O(\varepsilon_{\max})$, hence (by Lemma 3.5) a space complexity in $O^*(||\mathcal{I}||\varepsilon_{\max})$ and (using the traditional algorithm of Schönhage and Strassen) a time complexity in $O^*(||\mathcal{I}||\varepsilon_{\max} + \varepsilon_{\max} \log \varepsilon_{\max} \log\log \varepsilon_{\max}) = O^*(\varepsilon_{\max}(||\mathcal{I}|| + \log \varepsilon_{\max} \log\log \varepsilon_{\max})) = O^*(||\mathcal{I}||\varepsilon_{\max})$. □

Lemma 3.7. For a given $\varepsilon_{\max}$, each vector $\vec{N}^{=}(\vec{B}, J')$ indexed by $\varepsilon \leqslant \varepsilon_{max}$ can be computed in $O^*(2^{|J'|}||\mathcal{I}||\varepsilon_{\max})$ time and $O^*(||\mathcal{I}||\varepsilon_{\max})$ space.

Proof. Computing $\vec{N}^{=}(\vec{B}, J')$ requires $2^{|J'|}$ computations of $\vec{N}^{=}_{X}(\vec{B})$ by Inclusion-Exclusion. □

Proposition 3.8. The optimum objective value γ^{opt} (and $\varepsilon_{\max}$) can be computed in $O^*(2^n||\mathcal{I}||\gamma^{opt})$ time and $O^*(||\mathcal{I}||\gamma^{opt})$ space.

Proof. Computing $\varepsilon_{\max}$ and γ^{opt} using Algorithm 3.5 p. 87 consists in trying the successive powers of 2 for $\varepsilon_{\max}$, *i.e.* $\varepsilon_{\max} = 2^0, 2^1, 2^2, \ldots, 2^k, \ldots, 2^\ell$, while a criterion equivalent to $\varepsilon_{\max} < \gamma^{opt}$ holds. So, we always have $\varepsilon_{\max} \leqslant 2^\ell < 2\gamma^{opt}$, so $\varepsilon_{\max} = O(2^\ell) = O(\gamma^{opt})$. At each iteration k, we compute $\vec{N}^{=}(\vec{B}, J')$ with $\varepsilon \leqslant \varepsilon_{max} = 2^k$ and $J' = \{1, \ldots, n\}$. So, by Lemma 3.7, the k-th stage is in $O^*(2^n||\mathcal{I}||2^k)$ time and $O^*(||\mathcal{I}||2^k)$ space. The overall space complexity is the maximum of the space complexities at each iteration, in $O^*(||\mathcal{I}||2^\ell) = O^*(||\mathcal{I}||\gamma^{opt})$. The overall time complexity is the sum of the time complexities at each iteration, *i.e.* this time complexity is in $O^*(2^n||\mathcal{I}||(2^0+2^1+\ldots+2^\ell)) = O^*(2^n||\mathcal{I}||2 \cdot 2^\ell) = O^*(2^n||\mathcal{I}||\gamma^{opt})$. □

Proposition 3.9. An optimal schedule can be computed in $O^*(2^n||\mathcal{I}||\gamma^{opt})$ time and $O^*(||\mathcal{I}||\gamma^{opt})$ space.

Proof. Once γ^{opt} has been computed, an operation in $O^*(2^n||\mathcal{I}||\gamma^{opt})$ time and $O^*(||\mathcal{I}||\gamma^{opt})$ space, computation of an optimal schedule by Algorithm 3.2 p. 77 requires to compute a polynomial number of vectors $\vec{N}^{=}(\vec{B}, J')$ with $\varepsilon \leqslant \varepsilon_{max} = \gamma^{opt}$, each operation being in $O^*(2^n||\mathcal{I}||\gamma^{opt})$ time and $O^*(||\mathcal{I}||\gamma^{opt})$ space. □

3.4 Conclusions

In this chapter, we studied exact algorithms to minimize objective functions defined as a maximum or total regular cost, for unrelated parallel machine scheduling problems with release dates and deadlines, *i.e.* the $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ and $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ problems. From a theoretical point of view, we focus on bounding worst-case time and space complexities.

The best general complexity bounds proved so far are due to Lenté et al. [50]. They are in $O^*(3^n)$ time and $O^*(2^n)$ space in case of identical parallel machines without release times nor deadlines, *i.e.* for the $P||f_{\max}$ and $P||\sum_j f_j$ problems. Jansen et al. [40] get a more general but less precise result: for the general $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ and $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ problems, they prove that there exists an algorithm with time and space worst-case complexities in $O^*(c^n||\mathcal{I}||^{O(1)})$ for some constant c.

We described an Inclusion-Exclusion based algorithm for the general $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ and $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ problems, using dynamic programming for enumerations. In addition to the techniques usually associated with Inclusion-Exclusion, such as self-reducibility or the

relaxation of the coverage constraint, we transform relaxed problems with m parallel machines into m independent relaxed single-machine problems. This transformation is very important because it reduces the pseudopolynomial factor of the complexities from $||\mathcal{I}||^m$ to $||\mathcal{I}||$.

In addition to these techniques, we take care to limit the overhead due to the calculations on the cardinals of sets involved in the Inclusion-Exclusion formula. If we are not careful, these cardinals can be of the order of $2^{||\mathcal{I}||}$ and multiply the complexity by $||\mathcal{I}||$. By a simple technique, we guarantee that cardinals are bounded by n^n and, thus, we only multiply the complexity by a polynomial factor.

While the algorithm we describe to cope with the general $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem generalizes the ideas initiated in the article of Karp [42], the algorithm we describe to cope with the general $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ problem uses a more elaborate combinatorial technique, involving generating series and convolutions.

Finally, our main result is to describe an algorithm with a moderately exponential time complexity, comparable to that of Jansen et al., yet more precise, and especially with a worst-case space complexity bound which is only pseudopolynomial. We described an algorithm with a worst-case space complexity bound in $O^*(||\mathcal{I}||)$ for the $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem and in $O^*(||\mathcal{I}||\gamma^{opt})$ for the $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ problem, and with a worst-case time complexity bound in $O^*(2^n||\mathcal{I}||)$ for the $R|r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem and in $O^*(2^n||\mathcal{I}||\gamma^{opt})$ for the $R|r_{ij}, \tilde{d}_{ij}|\sum f_{ij}$ problem, where γ^{opt}, the optimum objective value, is itself a pseudopolynomial factor.

Contributions

- From a theoretical point of view, we demonstrate that a very general class of parallel machine problems with regular objectives can be solved with an exponential time worst-case complexity and a pseudopolynomial space worst-case complexity.
- The overhead induced by the computations on large cardinals involved in Inclusion-Exclusion is often neglected. We take this overhead into account and provide a simple method to guarantee that it is polynomially bounded.
- These results were the subject of two presentations: one at the ROADEF2021 [67] french conference, another at the PMS2022 [69] international conference.
- These results are published in the Journal of Combinatorial Optimization [72].

Chapter 4

Flowshop scheduling

Topics

- We deal with permutation flowshop scheduling problems in presence of job release dates and deadlines, to minimize general regular maximum and sum objective functions.
- Using Inclusion-Exclusion, we provide a generic algorithm for solving these problems, while running with moderate exponential-time and pseudopolynomial-space worst-case complexities.
- In order to cope with precedence constraints between jobs, we propose a hybrid algorithm in which Inclusion-Exclusion interacts nicely with dynamic programming across subsets.
- As a first step towards the solution of non-permutation flowshop problems, we study the application of Inclusion-Exclusion on a per-operation basis instead of a per-job basis.

In this chapter, we are interested in minimizing the maximal or total cost of jobs in a permutation flowshop schedule, in presence of operation dependent release times and deadlines. Using Graham's notations, these problems are denoted by $F|prmu, r_{ij}, \tilde{d}_{ij}|f_{\max}$ or $F|prmu, r_{ij}, \tilde{d}_{ij}|\sum f_j$. Graham's notations have been described in section 1.1.1 and flowshop problems have been described in section 1.1.2 and in Example 1.5 p. 30, but hereafter we recall the precise notations we will use throughout this chapter.

In the $F|prmu, r_{ij}, \tilde{d}_{ij}|f_{\max}$ and $F|prmu, r_{ij}, \tilde{d}_{ij}|\sum f_j$ problems, there are n jobs to be scheduled on m machines. Each job must be processed on machines 1 to m, in this order. Each machine can process only one job at a time. All machines must process jobs in the same order, and a schedule is essentially defined by this order. We denote by O_{ij}, $i \in \{1, \ldots, n\}$, $j \in \{1, \ldots, m\}$, the operation of job j on machine i, which has a non-negative integer processing time p_{ij}. For any given schedule S, we define $C_{ij}(S)$ as the completion time of O_{ij} in S. To each job j is assigned a cost, obtained by computing a regular *i.e.* non-decreasing cost function f_j of the completion time $C_{mj}(S)$ of the last

operation of j. Then, the objective function is defined as $\gamma(S) = \max_{1 \leqslant j \leqslant n} f_j(C_{mj}(S))$ or $\gamma(S) = \sum_{1 \leqslant j \leqslant n} f_j(C_{mj}(S))$. The aim is to find an optimal solution which minimizes the objective function. Whenever there is no ambiguity we will drop off the mention to S in the notations.

We define $J = \{1, \ldots, n\}$ as the set of jobs to be scheduled. An instance $\mathcal{I}$ of the $F|prmu, r_{ij}, \tilde{d}_{ij}|f_{\max}$ or $F|prmu, r_{ij}, \tilde{d}_{ij}|\sum f_j$ problem is of the form $\mathcal{I} = (p_{ij}, r_{ij}, \tilde{d}_{ij}, f_j)_{1 \leqslant i \leqslant m,\, j \in J}$.

In the literature, the most studied objective is the makespan, *i.e.* the maximum completion time. The $F|prmu|C_{\max}$ problem is strongly NP-hard (Garey et al. [28]). The brute force algorithm obviously has a time complexity in $O^*(n!)$. The very specific sub-problem $F2|prmu|C_{\max}$ can be polynomially solved by Johnson's rule [41], while the $F3|prmu|C_{\max}$ (Garey et al. [28]), and even the 2-machine problem with constraints, as $F2|chains|C_{\max}$ or $F2|r_j, \tilde{d}_j|C_{\max}$, are strongly NP-hard (Lenstra et al. [48]).

Unless $\mathcal{P} = \mathcal{NP}$, this implies that there are no polynomial time, and even no pseudopolynomial time, exact algorithms for the $F|prmu|C_{\max}$ problem, hence for the $F|prmu|f_{\max}$ problem. To get around this issue, numerous heuristics have been proposed in the literature to produce approximate solutions. We refer to Framinan et al. [24] for a survey on heuristics for the flowshop scheduling problem.

Williamson et al. [89] proved that no polynomial time heuristic for the flowshop problem with makespan objective may have a better approximation ratio than 5/4. So, it is necessary to use a super-polynomial algorithm in order to get a guaranteed optimal solution of the problem, or even an algorithm with a worst-case ratio smaller than 5/4. Moreover, under the Exponential Time Hypothesis introduced by Impagliazzo and Paturi [38], Jansen et al. [40] and independently Shang [78] showed that the flowshop problem cannot be solved by any sub-exponential algorithm, thus justifying the design of exact exponential algorithms.

As we have seen in section 1.3.3, Branch-and-Bound based algorithms solving permutation flowshop problems are very popular. While efficient in practice, they often have a worst-case time complexity bound comparable to the $O^*(n!)$ complexity of the brute-force algorithm. From a theoretical point of view, few algorithms have been proposed in order to provide better worst-case complexity bounds. Jansen et al. [40] described a very general algorithm class, in which they apply a dynamic programming technique on (unordered) sets of jobs. They get time and space worst-case complexities with respect to the number of operations $m{\cdot}n$, which translates into $2^{O(n)}||\mathcal{I}||^{O(1)}$ for each fixed number of machines. Shang et al. [80] gave a more precise result in the particular case of the $F3|prmu|C_{max}$ problem, by a fine analysis of the number of critical paths in a schedule. They obtained time and space complexities in $O^*(2^n||\mathcal{I}||)$.

Our main contribution is an algorithm which, for any fixed number of machines, runs with a moderate exponential worst-case time complexity and requires only pseudopolynomial space to minimize the general $\gamma = f_{max}$ or $\gamma = \sum f_j$ objective functions. More precisely, for the $F|prmu, r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem, the time complexity bound is in $O^*(2^n||\mathcal{I}||^m)$ and

the space complexity bound is in $O^*(||\mathcal{I}||^m)$. For the $F|prmu, r_{ij}, \tilde{d}_{ij}|\sum f_j$ problem, these bounds have to be multiplied by the optimum objective value γ^{opt}, which is a pseudopolynomial factor.

We also describe an extension of this algorithm to the $F|prmu, prec, r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem. For general job precedences, the space bound turns out to be exponential, but the time bound is unchanged. The same extension applies to the $F|prmu, prec, r_{ij}, \tilde{d}_{ij}|\sum f_j$ problem with a time bound multiplied by the pseudopolynomial factor γ^{opt}.

As as we have experienced in chapter 3, the solution of the scheduling problems we consider comprises a rather mechanical phase and a more specific phase. In section 4.1, we describe how to apply the Inclusion-Exclusion principle to permutation flowshop problems. In sections 4.2 and 4.3, we apply these principles to the $F|prmu, r_{ij}, \tilde{d}_{ij}|f_{\max}$ and $F|prmu, r_{ij}, \tilde{d}_{ij}|\sum f_j$ problems. In section 4.4, we discuss an adaptation of the algorithm to the case of job precedence constraints. Finally, in section 4.5, we adapt the Inclusion-Exclusion principle to manage operations instead of jobs. This adaptation is a first step towards the solution of non-permutation flowshop problems.

4.1 Principles

In this chapter we consider a large class of permutation flowshop problems whose objectives are regular. As we have seen in section 1.1.5, semi-active schedules form a dominant set, and it is possible to represent unambiguously a semi-active schedule S by a list of jobs $(j_k)_{1\leqslant k\leqslant n}$ indexed by chronological order k. Such a list is a permutation of jobs, *i.e.* $S \in \mathfrak{S}_n$.

Before we solve the $F|prmu, r_{ij}, \tilde{d}_{ij}|\sum f_j$ and $F|prmu, r_{ij}, \tilde{d}_{ij}|f_{\max}$ problems, we have to tackle an issue: due to the presence of deadlines, it may happen that there is no solution. To determine whether there exists a solution, we remove the costs and replace them with the trivial, null costs, *i.e.* we take as objective function $\gamma = 0 = \max_j 0 = \sum_j 0$. So, we consider the fake optimization problem $\mathcal{P}^0_{opt} = F|prmu, r_{ij}, \tilde{d}_{ij}|$-, which has the same set of admissible schedules as the $F|prmu, r_{ij}, \tilde{d}_{ij}|\sum f_j$ and $F|prmu, r_{ij}, \tilde{d}_{ij}|f_{\max}$ problems. As we saw in section 2.1.1, an optimization problem reduces to a decision problem. To the fake optimization problem $\mathcal{P}^0_{opt}$ is associated a decision problem $\mathcal{P}^0_{dec}(\mathcal{I}, \varepsilon)$, which we will solve in section 4.2 or 4.3 p. 98. Then, $F|prmu, r_{ij}, \tilde{d}_{ij}|\sum f_j$ and $F|prmu, r_{ij}, \tilde{d}_{ij}|f_{\max}$ have a solution if and only if $\mathcal{P}^0_{opt}$ has a solution, if and only if $\mathcal{P}^0_{dec}(\mathcal{I}, 0)$ is true.

4.1.1 Self-reducibility

Following Remark 2.5 p. 49, we study the effect of appending a job to a prefix. Figure 4.1 p. 94 shows a permutation flowshop schedule with a job j, the completion front $\vec{B}$ before j *i.e.* the completion front of the previous job, and the completion front $\vec{C}$ of j:

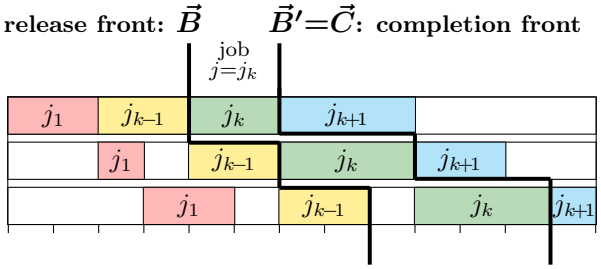


Figure 4.1: Time fronts in a permutation schedule

Given a job j, we consider the time front $\vec{C}_j = (C_{1j}, \ldots, C_{mj})$ of completion times of all operations of j. The cost of a job is computed according to the completion time of its last operation so, as a notation we extend cost functions from completion times to completion fronts by defining:

$$f_j(\vec{C}) = f_j(C_{mj}) \tag{4.1}$$

These completion times are also lower bounds of the starting times, *i.e.* release dates, of operations of all jobs scheduled after j. As in section 3.1.1, we denote such a time front by $\vec{B}$ and call it a release bound front. Considering a release bound front $\vec{B} = (B_1, \ldots, B_m)$ before a job j is scheduled amounts to replace the release dates of its operations r_{ij} by $\max(B_i, r_{ij})$.

We need an upper bound, say $B_{\max}$, for the components of all possible release bound fronts. Since a release bound front is also a completion time front, we just have to take an upper bound of the makespan of any schedule (*i.e.* $\forall S, C_{\max}(S) \leqslant B_{\max}$). So, we define $B_{\max}$ by the following formula, which is a very loose upper bound, where we add the maximum release time and the sum of all processing times:

$$B_{\max} \;=\; \max_{i,j} r_{ij} \;+\; \sum_{i,j} p_{ij} \tag{4.2}$$

Notice that $B_{\max} = O^*(||\mathcal{I}||)$. This will be useful for bounding the number of states of dynamic programming schemes.

We define an operator $\bullet$ such that, given a completion front $\vec{B}$ and a job j, $\vec{B}' = \vec{B} \bullet j$ is the completion front we get by processing job j after the release bound front $\vec{B}$. Due to the precedences between operations implied by the permutation flowshop problem, we have, with the convention that $B'_0 = 0$:

$$B'_i = \max(B'_{i-1}, B_i) + p_{i,j} \qquad \forall\, i = 1, \ldots, m \tag{4.3}$$

Assume a schedule $S = (j_1, \ldots, j_n)$ is given and let us define $\vec{B}_{j_k}$ as the release bound front of j_k, *i.e.* the completion front of j_{k-1}, with the convention that $\vec{B}_{j_1}$ contains the initial times. The completion front $\vec{B}_{j_{k+1}}$ is uniquely determined by the completion front $\vec{B}_{j_k}$, and we have:

$$\vec{B}_{j_1} \;=\; \vec{0} \;=\; (0, \ldots, 0) \tag{4.4}$$

$$\vec{B}_{j_{k+1}} = \vec{B}_{j_k} \bullet j_k \qquad \text{for } k \geqslant 1 \tag{4.5}$$

The completion front of the whole schedule $S = (j_1, \ldots, j_n)$ is the completion front of its last job, i.e:

$$\vec{C}((j_1, \ldots, j_n)) \;=\; \vec{C}_{j_n} \;=\; \vec{B}_{j_{n+1}} \;=\; \vec{0} \bullet i_1 \bullet \ldots \bullet i_n \tag{4.6}$$

This formula straightforwardly extends to any schedule prefix $\pi = (j_1, \ldots, j_h)$ instead of $S = (j_1, \ldots, j_n)$. A flowshop scheduling problem is made self-reducible by considering the completion times of any prefix π as release bounds of operations of any job scheduled after π. We will apply this principle to derive precise formulas to implement $\mathcal{P}_{dec}$ in sections 4.2 and 4.3 p. 98. Then, the optimum objective value γ^{opt} is derived from Algorithm 2.2 p. 48 and an optimal schedule is derived from Algorithm 2.3 p. 48, as explained in section 2.1.1.

Notice that the $\bullet$ operator is non-decreasing with respect to release bound fronts (or equivalently completion fronts): we define the (partial) chronological order on release bound fronts as the chronological order on each machine, *i.e.* $\vec{B} \leqslant \vec{B}' \iff \forall\, i \in \{1, \ldots, m\},\ B_i \leqslant B'_i$. Then, appending a job j preserves this order, *i.e.* $\vec{B} \leqslant \vec{B}' \Longrightarrow \vec{B} \bullet j \leqslant \vec{B}' \bullet j$. Actually, any variant of the $F|prmu|f_{\max}$ or $F|prmu|\sum f_j$ problems for which a non-decreasing $\bullet$ operator exists can be solved by the algorithms proposed in this chapter.

4.1.2 Comparison with dynamic programming across subsets

As stated in Remark 2.5 p. 49, Equations (4.4) p. 94 and (4.5) p. 94 enable to derive not only self-reducibility, but also a dynamic programming scheme and an Inclusion-Exclusion based algorithm. As an example, we focus on dynamic programming schemes across subsets to solve the $F|prmu|f_{\max}$ and $F|prmu|\sum f_j$ problems (with no release times nor deadlines). As the objective value of a schedule $S = (j_1, \ldots, j_n)$ is the maximum or total cost, we have:

$$f_{\max} = \max(f_{j_1}(\vec{0} \bullet j_1),\ \ldots,\ f_{j_n}(\vec{0} \bullet j_1 \bullet \ldots \bullet j_n)) \tag{4.7}$$

$$\sum f_j = f_{j_1}(\vec{0} \bullet j_1) + \ldots + f_{j_n}(\vec{0} \bullet j_1 \bullet \ldots \bullet j_n) \tag{4.8}$$

Given a release bound front $\vec{B}$ and a job subset $J \subset \{1, \ldots, n\}$, we define $opt[\vec{B}, J]$ as the minimal objective of permutation schedules starting from $\vec{B}$ and using exactly once the jobs of J. We have, for the $F|prmu|f_{\max}$ problem:

$$opt[\vec{B}, \varnothing] = 0 \tag{4.9}$$

$$opt[\vec{B}, J] = \min_{j \in J} \left(\max(f_j(\vec{B} \bullet j), opt[\vec{B} \bullet j, J \backslash \{j\}]) \right) \qquad \forall\, J \neq \varnothing \tag{4.10}$$

and, for the $F|prmu|\sum f_j$ problem:

$$opt[\vec{B}, \varnothing] = 0 \tag{4.11}$$

$$opt[\vec{B}, J] = \min_{j \in J} \left(f_j(\vec{B} \bullet j) + opt[\vec{B} \bullet j, J \backslash \{j\}] \right) \qquad \forall\, J \neq \varnothing \tag{4.12}$$

Then, for an instance with $|J| = n$ jobs, the minimal objective of a permutation schedule using the jobs of J is $opt[\vec{0}, J]$, and a corresponding optimal schedule can be found by computing the backtrace of the dynamic programming scheme. This scheme uses simple

equations, but it explores completion fronts and subsets $J' \subset J$, with $|J| = n$, thus leading to an algorithm with a time and space complexity in $O^*(2^n||\mathcal{I}||^m)$. An enhancement of this dynamic programming scheme based on Pareto fronts is proposed by Shang et al. [80].

Once again, as we have seen in section 2.2 and as we will see in sections 4.2 and 4.3 p. 98, the interest of Inclusion-Exclusion is to lead to an algorithm with a pseudo-polynomial worst-case space complexity bound while preserving the worst-case time complexity bound and the simplicity of the equations of a dynamic programming scheme across subsets.

4.2 Minimizing a maximum cost

In this section, we focus on the $\mathcal{P}_{opt} = F|prmu, r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem. In order to solve it, it is sufficient to solve, for each prefix π and objective threshold ε, the decision problem $\mathcal{P}_{dec}(\mathcal{I}, \pi, \varepsilon)$ defined by Equation (2.1) p. 47. We define the number $N(\vec{B}, J', \varepsilon)$ of admissible schedules formed of jobs in $J' \subset J$, starting from bound front $\vec{B}$ and with objective at most ε. As shown in Figure 4.1 p. 94, imposing a prefix amounts to imposing that the remaining jobs are outside the prefix and have release time bounds B_i equal to the completion times of the prefix on each machine, as defined by Equation (4.6) p. 95. We derive:

$$\mathcal{P}_{dec}\Big(\mathcal{I},\ (\pi_1, \ldots, \pi_h),\ \varepsilon\Big) \iff N\left(\begin{pmatrix} C_{1,\pi_h} \\ \cdots \\ C_{m,\pi_h} \end{pmatrix},\ J \setminus \pi,\ \varepsilon\right) > 0 \tag{4.13}$$

4.2.1 Inclusion-Exclusion

The strict schedules we consider are permutations of jobs, *i.e.* elements of $\mathfrak{S}_{J'}$, viewed as job lists. So, the job lists we consider are elements of $J'^* = J'^{n'}$, where $n' = |J'|$, We relax the coverage constraint, *i.e.* we allow job lists with missing or duplicate jobs. The set of admissible relaxed schedules $\mathscr{R} \subset J'^*$ is composed of lists of exactly n' jobs, containing potentially duplicate or missing jobs, with no deadline violation, and with objective, *i.e.* maximum cost $\gamma = \max f_j$, at most ε. We derive:

$$\mathscr{R} = \left\{(j_1, \ldots, j_{n'}) \;\middle|\; \forall i, k, C_{i,j_k} \leqslant \tilde{d}_{i,j_k} \ \wedge\ \forall k, f_{j_k}(C_{m,j_k}) \leqslant \varepsilon\right\} \tag{4.14}$$

In our case, permutations are exactly covering lists, so the number $N(\vec{B}, J', \varepsilon)$ of (strict) admissible schedules is also the number of admissible covering relaxed schedules. For a job subset $X \subset J'$ and a threshold objective value ε, we define $N_{X,\varepsilon}[\vec{B}, \ell]$ as the number of admissible relaxed schedules composed of ℓ jobs, all in X, starting from $\vec{B}$, and with objective value at most ε. Then, we derive from Formula (2.17) p. 53:

$$N(\vec{B}, J', \varepsilon) = \sum_{X \subset J'} (-1)^{|J'|-|X|} N_{X,\varepsilon}[\vec{B}, n'] \tag{4.15}$$

We now just have to compute $N_{X,\varepsilon}[\vec{B}, \ell]$ by dynamic programming. For $\ell = 0$ there is only one schedule, the empty one, and it is admissible. For $\ell > 0$, we try each possible job j in the first position. There are admissible schedules to count if j does not violate its

deadlines and if the cost of job j is not larger than ε, in which case the completion times of j become the release bounds of the other jobs, and we recursively count the possible suffixes. We derive:

$$N_{X,\varepsilon}[\vec{B}, 0] = 1 \tag{4.16}$$

$$N_{X,\varepsilon}[\vec{B}, \ell] = \sum_{\substack{j \in X \\ \forall\, i, C_{ij} \leqslant \tilde{d}_{ij} \\ f_j(C_{mj}) \leqslant \varepsilon \\ \text{where } \vec{C}_j = \vec{B} \bullet j}} N_{X,\varepsilon}[\vec{C}_j, \ell - 1] \qquad \text{for } \ell \neq 0 \tag{4.17}$$

4.2.2 Worst-case complexity bounds

Lemma 4.1. Each value $N(\vec{B}, J', \varepsilon)$ can be computed, for every m, in $O^*(2^n ||\mathcal{I}||^m)$ time and $O^*(||\mathcal{I}||^m)$ space.

Proof. Each state involved in the dynamic programming scheme to compute $N_{X,\varepsilon}[\vec{B}, \ell]$ is formed with a number of jobs $\ell \leqslant n' \leqslant n$ and m components $B_i \leqslant B_{\max} = O^*(||\mathcal{I}||)$, so there are at most $(n+1)(||\mathcal{I}||+1)^m = O^*(||\mathcal{I}||^m)$ states. Consequently, it takes $O^*(||\mathcal{I}||^m)$ time and space to compute each $N_{X,\varepsilon}[\vec{B}, \ell]$ by dynamic programming. Due to the Inclusion-Exclusion formula, $N(\vec{B}, J', \varepsilon)$ is obtained by summing $2^{n'} \leqslant 2^n$ terms $N_{X,\varepsilon}[\vec{B}, \ell]$, giving the result. □

Remark 4.2. The numbers computed inside the dynamic programming scheme are potentially large cardinals, so their computation time and space is not constant. But their values are bounded by $n||\mathcal{I}||^m$, so their computation induces an overhead of $O(\log n + m \log ||\mathcal{I}||)$, and this polynomial overhead is hidden by the O^* notation.

Proposition 4.3. The optimal objective value can be computed, for every m, in $O^*(2^n ||\mathcal{I}||^m)$ time and $O^*(||\mathcal{I}||^m)$ space.

Proof. As shown in section 2.1.1, Algorithms 2.2 p. 48 and 2.3 p. 48 use a polynomial number of calls to $N(\vec{B}, J', \varepsilon)$ to compute an optimal schedule. □

4.2.3 An enhancement for makespan minimization

For the specific $F|prmu, r_{ij}, \tilde{d}_{ij}|C_{\max}$ problem, *i.e.* makespan minimization, we can avoid the binary search carried out by Algorithm 2.2 p. 48 and save a polynomial factor $\log(||\mathcal{I}||)$ in the time complexity bound. Given a job set $X \subset J$, we define, for each number of jobs k and completion front $\vec{C}$, $N^*_{X,k}[\vec{C}]$ as the number of relaxed schedules with k jobs, all in X, whose completion front is $\vec{C}$. We also define, for each objective value ε, $N^*_X[\varepsilon]$ as the number of relaxed schedules with n jobs, all in X, whose objective $C_{\max}$ is exactly ε.

These definitions are implemented as association tables (from keys to values) computed by Algorithm 4.2 p. 98. There is one schedule with 0 job and its completion front is $\vec{0}$. All schedules with $(k-1)$ jobs and completion front $\vec{C}$ followed by a job j form schedules with k jobs and completion front $(\vec{C} \bullet j)$, so their number $N^*_{X,k-1}[\vec{C}]$ adds up to $N^*_{X,k}[\vec{C} \bullet j]$,

provided that $(\vec{C} \bullet j)$ is compatible with the deadlines of j. Likewise, all schedules with the same completion front $\vec{C}$ share the same objective $C_{\max} = C_m$, so their number $N^*_{X,n}[\vec{C}]$ adds up to $N^*_X[C_m]$. Notice that this reasoning does not hold for a general $f_{\max}$ objective.

Algorithm 4.2: Computation of the N^*_X table

Procedure *ComputeTables(X)*:
  // Values are implicitly null for undefined keys
  $N^*_{X,0}[\vec{0}] \leftarrow 1$
  for $k = 1$ *to* n **do**
    for each $\vec{C}$ *key of* $N^*_{X,k-1}$ **do**
      for each $j \in X$ **do**
        if $\forall i, (\vec{C} \bullet j)_i \leqslant \tilde{d}_{ij}$ **then**
          $N^*_{X,k}[\vec{C} \bullet j] \leftarrow N^*_{X,k}[\vec{C} \bullet j] + N^*_{X,k-1}[\vec{C}]$
  for each $\vec{C}$ *key of* $N^*_{X,n}$ **do**
    $N^*_X[C_m] \leftarrow N^*_X[C_m] + N^*_{X,n}[\vec{C}]$

We now define, for each objective value ε, $N^*[\varepsilon]$ as the number of covering schedules with n jobs, whose objective $C_{\max}$ is exactly ε. It can be computed by applying Inclusion-Exclusion on tables N^*_X considered as vectors indexed by ε, *i.e.* $N^* = \sum_{X \subset J} (-1)^{|J|-|X|} N^*_X$. Finally, the optimum objective is $C^{\text{opt}}_{\max} = \min\{\varepsilon \in \mathbb{N} \mid N^*[\varepsilon] > 0\}$.

4.3 Minimizing a total cost

In this section, we focus on the $\mathcal{P}_{opt} = F|prmu, r_{ij}, \tilde{d}_{ij}| \sum f_j$ problem. In order to solve it, it is sufficient to solve, for each prefix π and objective threshold ε, the decision problem $\mathcal{P}_{dec}(\mathcal{I}, \pi, \varepsilon)$ defined by Equation (2.1) p. 47. We denote by $N(\vec{B}, J', \varepsilon)$ the number of admissible schedules formed of jobs of J', starting from $\vec{B}$ and with objective at most ε. As shown in Figure 4.1 p. 94, imposing a prefix amounts to imposing that the remaining jobs are outside the prefix and have release time bounds B_i equal to the completion times of the prefix on each machine, as defined by Equation (4.6) p. 95. Notice that, because the objective is a sum of elementary costs, the jobs of the prefix consume a part of the total available objective ε and this part has to be subtracted from ε. We derive:

$$\mathcal{P}_{dec}\Big(\mathcal{I},\ (\pi_1, \ldots, \pi_h),\ \varepsilon\Big) \iff N\left(\begin{pmatrix} C_{1,\pi_h} \\ \cdots \\ C_{m,\pi_h} \end{pmatrix},\ J \setminus \pi,\ \varepsilon - \sum_{k=1}^{h} f_{\pi_k}(C_{m,\pi_k})\right) > 0 \qquad (4.18)$$

4.3.1 Inclusion-Exclusion

The strict schedules we consider are permutations of jobs, *i.e.* elements of $\mathfrak{S}_{J'}$, viewed as job lists. So, the job lists we consider are elements of $J'^* = J'^{n'}$, where $n' = |J'|$, We relax the coverage constraint, *i.e.* we allow job lists with missing or duplicate jobs. The set of admissible relaxed schedules $\mathscr{R} \subset J'^*$ is composed of lists of exactly n' jobs, containing potentially duplicate or missing jobs, with no deadline violation, and with objective, *i.e.* total cost $\gamma = \sum_j f_j$, at most ε. We derive:

$$\mathscr{R} = \left\{ (j_1, \ldots, j_{n'}) \;\middle|\; \forall\, i,k, C_{i,j_k} \leqslant \tilde{d}_{i,j_k} \;\wedge\; \sum_{k=1}^{n'} f_{j_k}(C_{m,j_k}) \leqslant \varepsilon \right\} \tag{4.19}$$

In our case, permutations are exactly covering lists, so the number $N(\vec{B}, J', \varepsilon)$ of (strict) admissible schedules is also the number of admissible covering relaxed schedules. For a job subset $X \subset J'$ and a threshold objective value ε, we define $N_X[\vec{B}, \ell, \varepsilon]$ as the number of admissible relaxed schedules composed of ℓ jobs, all in X, starting from $\vec{B}$, and with objective value at most ε. Then, we derive from Formula (2.17) p. 53:

$$N(\vec{B}, J', \varepsilon) = \sum_{X \subset J'} (-1)^{|J'|-|X|} N_X[\vec{B}, n', \varepsilon] \tag{4.20}$$

We now just have to compute $N_X[\vec{B}, \ell, \varepsilon]$ by dynamic programming. For $\ell = 0$ there is only one schedule, the empty one, and it is admissible. For $\ell > 0$, we try each possible job j in the first position. There are admissible schedules to count if j does not violate its deadlines and if the cost of job j is not larger than ε, in which case this cost has to be subtracted from ε and the completion times of j become the release bounds of the other jobs, and we recursively count the possible suffixes. We derive:

$$N_X[\vec{B}, 0, \varepsilon] = \; 1 \tag{4.21}$$

$$N_X[\vec{B}, \ell, \varepsilon] = \sum_{\substack{j \in X \\ \forall\, i, C_{ij} \leqslant \tilde{d}_{ij} \\ f_j(C_{mj}) \leqslant \varepsilon \\ \text{where } \vec{C}_j = \vec{B} \bullet j}} N_X[\vec{C}_j, \ell - 1, \varepsilon - f_j(C_{mj})] \qquad \text{for } \ell \neq 0 \tag{4.22}$$

4.3.2 Computation of the optimal objective value

We could use the standard algorithm based on a binary search, *i.e.* Algorithm 2.2 p. 48, to compute the optimal objective value γ^{opt}. But this is not the best we can do, because, as the objective threshold ε is part of the dynamic programming state involved in Equation (4.22), computing a single value $N_X[\vec{B}, \ell, \varepsilon]$ is as costly as computing together all values $N_X[\vec{B}, \ell, \varepsilon']$ for $\varepsilon' \leqslant \varepsilon$.

As for the solution of the $R|r_{ij}, \tilde{d}_{ij}| \sum f_{ij}$ problem in section 3.3, we apply a vectorial version of Inclusion-Exclusion. Given a maximum objective threshold $\varepsilon_{\max}$, we define these vectors indexed by all $\varepsilon \leqslant \varepsilon_{\max}$:

$$\vec{N}(\vec{B}, J', \varepsilon_{\max}) = \big(N(\vec{B}, J', 0), \ldots, N(\vec{B}, J', \varepsilon_{\max})\big) \tag{4.23}$$

$$\vec{N}_X(\vec{B}, \ell, \varepsilon_{\max}) = \big(N_X(\vec{B}, \ell, 0), \ldots, N_X(\vec{B}, \ell, \varepsilon_{\max})\big) \tag{4.24}$$

and we derive from Formula (2.17) p. 53:

$$\vec{N}(\vec{B}, J', \varepsilon_{\max}) = \sum_{X \subset J'} (-1)^{|J'|-|X|} \vec{N}_X(\vec{B}, n', \varepsilon_{\max}) \tag{4.25}$$

We can actually compute both $\varepsilon_{\max}$ and γ^{opt} as follows: choose an arbitrary $\varepsilon_{\max}$ and compute $\vec{N} = \vec{N}(\vec{B}, J', \varepsilon_{\max})$. Then, the minimum objective is the lowest ε, if it exists, such that $\vec{N}_\varepsilon = N(\vec{B}, J', \varepsilon)$ is not null. If no such ε exists, this means that $\varepsilon_{\max}$ is too small and must be increased. During this computation, we do not need any prefix π, because such a prefix is only required to compute an optimal schedule. So, we take $\vec{B} = \vec{0}$, *i.e.* no release bounds, and $J' = \{1, \ldots, n\}$, *i.e.* the whole set of jobs. Similarly to Algorithm 3.5 p. 87, we propose Algorithm 4.3 to compute the optimum total cost objective value:

Algorithm 4.3: Computation of the optimum objective value for a total cost objective

Function *OptimumTotalCost*:
- $\vec{B} \leftarrow \vec{0}$
- $J' \leftarrow \{1, \ldots, n\}$
- $\varepsilon_{\max} \leftarrow 1$
- $\vec{N} \leftarrow \vec{N}(\vec{B}, J', \varepsilon_{\max})$
- **while** $\vec{N} = \vec{0}$ **do**
  - $\varepsilon_{\max} \leftarrow 2\ \varepsilon_{\max}$
  - $\vec{N} \leftarrow \vec{N}(\vec{B}, J', \varepsilon_{\max})$
- $\gamma^{opt} \leftarrow \min\{0 \leqslant \varepsilon \leqslant \varepsilon_{\max} \mid \vec{N}_\varepsilon > 0\}$
- **return** γ^{opt}

When γ^{opt} has been computed, an optimal solution can be derived from Algorithm 2.3 p. 48 and without any vectorial computation.

4.3.3 Worst-case complexity bounds

Lemma 4.4. Each vector $\vec{N}(\vec{B}, J', \varepsilon)$ and thus each single value $N(\vec{B}, J', \varepsilon)$ can be computed, for every m, in $O^*(2^n ||\mathcal{I}||^m \varepsilon)$ time and $O^*(||\mathcal{I}||^m \varepsilon)$ space.

Proof. Each state involved in the dynamic programming scheme to compute $N_X[\vec{B}, \ell, \varepsilon']$ is formed with a number of jobs $\ell \leqslant n' \leqslant n$, m components $B_i \leqslant B_{\max} = O^*(||\mathcal{I}||)$, and a threshold value $\varepsilon' \leqslant \varepsilon$, so there are at most $(n+1)(||\mathcal{I}||+1)^m \varepsilon = O^*(||\mathcal{I}||^m \varepsilon)$ states. Consequently it takes $O^*(||\mathcal{I}||^m \varepsilon)$ time and space to compute each vector $\vec{N}_X[\vec{B}, \ell, \varepsilon]$ by dynamic programming. Due to the Inclusion-Exclusion formula, $\vec{N}(\vec{B}, J', \varepsilon)$ is obtained by summing $2^{n'} \leqslant 2^n$ terms $\vec{N}_X[\vec{B}, \ell, \varepsilon]$, giving the result.

As stated in Remark 4.2 p. 97, the values of N are potentially large cardinals, but their computation induces a polynomial only overhead. □

Proposition 4.5. The optimal objective value can be computed, for every m, in $O^*(2^n||\mathcal{I}||^m\gamma^{opt})$ time and $O^*(||\mathcal{I}||^m\gamma^{opt})$ space.

Proof. Algorithm 4.3 p. 100 makes successive computations of $\vec{N}(\vec{B}, J', \varepsilon_{\max})$ with $\varepsilon_{\max} = 2^0, 2^1, 2^2, \ldots, 2^k, \ldots, 2^\ell$ and $\varepsilon_{\max} < 2\gamma^{opt}$. The k-th stage is in $O^*(2^n||\mathcal{I}||^m 2^k)$ time and $O^*(||\mathcal{I}||^m 2^k)$ space. The overall space complexity is the maximum of the space complexities at each iteration, in $O^*(||\mathcal{I}||^m 2^\ell) = O^*(||\mathcal{I}||^m\gamma^{opt})$. The overall time complexity is the sum of the time complexities at each iteration, *i.e.* it is in $O^*(2^n||\mathcal{I}||^m(2^0 + 2^1 + \ldots + 2^\ell)) = O^*(2^n||\mathcal{I}||^m 2 \cdot 2^\ell) = O^*(2^n||\mathcal{I}||^m\gamma^{opt})$.

Then, as shown in section 2.1.1, Algorithm 2.3 p. 48 uses a polynomial number of calls to $N(\vec{B}, J, \varepsilon)$, each in $O^*(2^n||\mathcal{I}||^m\gamma^{opt})$ time and $O^*(||\mathcal{I}||^m\gamma^{opt})$ space as $\varepsilon \leqslant \gamma^{opt}$, to compute an optimal schedule. □

4.4 Extension to the case of precedences between jobs

Job precedence constraints are defined by a partial order $\prec$ on jobs, with $j \prec j'$ meaning that job j must be scheduled before j'.

We focus on the $F|prmu, prec, r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem, *i.e.* the problem of minimizing a maximal cost under job precedences. Although its structure is similar to the one of the $F|prmu, r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem, precedences lead to the use of hybrid Inclusion-Exclusion formulas, which work in a traditional way while integrating dynamic programming across subsets of jobs. We provide below details on how it works.

In order to solve the $\mathcal{P}_{opt} = F|prmu, prec, r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem, we implement, for each instance $\mathcal{I}$, prefix π and objective threshold ε, the decision problem $\mathcal{P}_{dec}(\mathcal{I}, \pi, \varepsilon)$, defined by Equation (2.1) p. 47. We denote by $N(\vec{B}, J', \varepsilon)$ the number of admissible schedules formed of jobs of $J' \subset J$, starting from $\vec{B}$ and with objective at most ε. A schedule prefix can be completed into a schedule satisfying precedences if the prefix itself satisfies precedences and if no job appended to the prefix precedes any job of the prefix. We derive:

$$\mathcal{P}_{dec}\big(\mathcal{I}, (\pi_1, \ldots, \pi_h), \varepsilon\big) \iff \begin{cases} \forall\, k, k' \in \{1, \ldots, h\},\ \pi_k \prec \pi_{k'} \Longrightarrow k < k' & (4.26) \\ \forall\, j \in \pi,\ \ \forall\, j' \in J \setminus \pi,\ j \not\prec j' & (4.27) \\ N\left(\begin{pmatrix} C_{1,\,\pi_h} \\ \cdots \\ C_{m,\pi_h} \end{pmatrix},\ J \setminus \pi,\ \varepsilon \right) > 0 & (4.28) \end{cases}$$

These equations do not strictly conform to our definition 2.1 p. 47 of an implementation, but notice that the extra conditions (4.26) and (4.27) are computed in polynomial time.

4.4.1 Inclusion-Exclusion

We set $n' = |J'|$ and we relax the problem as in section 4.2. The set of admissible relaxed schedules $\mathscr{R} \subset {J'}^{n'}$ is composed of lists of exactly n' jobs but containing potentially duplicate or missing jobs, with no deadline violation, no precedence violation, and with an objective value at most ε. We derive:

$$\mathscr{R} = \left\{ (j_1, \ldots, j_{n'}) \;\middle|\; \forall i,k, C_{i,j_k} \leqslant \tilde{d}_{i,j_k} \;\wedge\; \forall k,k', j_k \prec j_{k'} \Rightarrow k < k' \;\wedge\; \forall k, f_{j_k}(C_{m,j_k}) \leqslant \varepsilon \right\} \tag{4.29}$$

For a job subset $X \subset J'$, we define $N_X(\vec{B}, J', \varepsilon)$ as the number of admissible relaxed schedules using only jobs of X and we derive from Formula (2.17) p. 53:

$$N(\vec{B}, J', \varepsilon) = \sum_{X \subset J'} (-1)^{|J'|-|X|} N_X(\vec{B}, J', \varepsilon) \tag{4.30}$$

In our case, this formula is oversized and we will simplify it.

We need some definitions to deal with precedences in the computation of $N_X(\vec{B}, J', \varepsilon)$. We only have to consider jobs of J', so we restrict the partial order $\prec$ to J'. Let $Prec(j) = \{j' \in J' \mid j' \prec j\}$ be the set of predecessors of j, and $Succ(j) = \{j' \in J' \mid j \prec j'\}$ be the set of successors of j. We define M as the set of maximal jobs for $\prec$, *i.e.* jobs without successors, and the set $J' \setminus M$ of non-maximal jobs is also the set of predecessors of other jobs. So, we have:

$$M = \{j \in J' \mid Succ(j) = \varnothing\} \tag{4.31}$$

$$J' \setminus M = \{j \in J' \mid Succ(j) \neq \varnothing\} = \bigcup_{j \in J'} Prec(j) \tag{4.32}$$

As stated in section 2.1.1, we proceed by forward dynamic programming: each dynamic programming state contains a history, derived from already scheduled jobs forming an implicit schedule prefix $\pi' = (j_1, \ldots, j_h)$, and recurrence equations model the effect of appending a job j_{h+1} to this prefix π'. All predecessors of j_{h+1} must belong to the prefix, otherwise there exists a predecessor of j_{h+1} scheduled outside the prefix, so after j_{h+1}, which is not admissible. Thus, we must have $Prec(j_{h+1}) \subset \pi'$. This relation forces us to keep track of jobs of π' which can be predecessors of other jobs, *i.e.* non-maximal jobs, or jobs of $\pi' \setminus M$. So, each state used in the dynamic programming scheme must incorporate the set $P \subset J' \setminus M$ of previously scheduled non-maximal jobs.

We introduce a hybrid dynamic programming scheme to compute $N_X(\vec{B}, J', \varepsilon)$. Each state contains a set $P \subset J' \setminus M$ of previously scheduled non-maximal jobs, conforming to the principle of dynamic programming across the subsets, and each state also contains usual data for scheduling: a release bound front $\vec{B}$ and a remaining length ℓ. We define the corresponding number of schedules $N_{X,\varepsilon}[P, \vec{B}, \ell]$ and we derive:

$$N_X(\vec{B}, J', \varepsilon) = N_{X,\varepsilon}[\varnothing, \vec{B}, n'] \tag{4.33}$$

Recurrence equations (4.34) and (4.35) mimic equations (4.16) p. 97 and (4.17) p. 97:

$$N_{X,\varepsilon}[P,\vec{B},0] = \begin{cases} 1 \text{ if } P = J' \setminus M, \\ 0 \text{ otherwise} \end{cases} \tag{4.34}$$

$$N_{X,\varepsilon}[P,\vec{B},\ell] = \sum_{\substack{j\in X \\ j\notin P \\ Prec(j)\subset P \\ \forall\, i, C_{ij}\leqslant \tilde{d}_{ij} \\ f_j(C_{mj})\leqslant\varepsilon \\ \text{where } \vec{C}_j = \vec{B}\bullet j}} N_{X,\varepsilon}[P\cup\{j\}\setminus M,\ \vec{C}_j,\ \ell-1] \qquad \text{for } \ell \neq 0 \tag{4.35}$$

In Equation (4.35), a job j is suitable if all its predecessors have already been scheduled, *i.e.* if $Prec(j) \subset P$, and if j itself has not already been scheduled, *i.e.* if $j \notin P$. Then, job j is added to the set of previously scheduled non-maximal jobs only if it is non-maximal, hence justifying the term $P \cup \{j\} \setminus M$ in Equation (4.35).

Notice that each set P computed during our dynamic programming scheme is a downward closed set of the poset $J' \setminus M$:

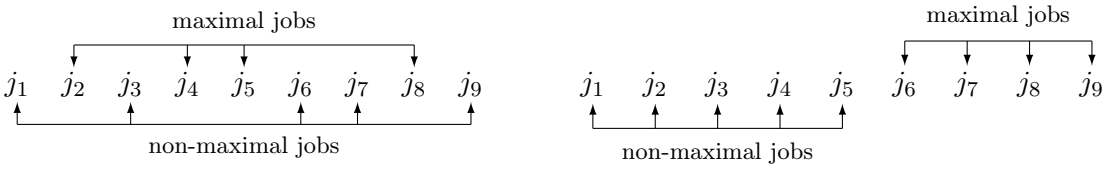
$$\forall\, j, j', \quad j' \prec j \quad \wedge \quad j \in P \implies j' \in P \tag{4.36}$$

Maximal jobs, *i.e.* jobs of M, are conceptually handled by Inclusion-Exclusion whereas non-maximal jobs are conceptually handled by dynamic programming across the subsets of $J' \setminus M$. The aim of Inclusion-Exclusion is to ensure the coverage of J'. We split this coverage constraint into two parts: M and $J' \setminus M$. The coverage of $J' \setminus M$ is ensured by Equation (4.34): an empty schedule suffix preceded by jobs of P can correspond to a covering schedule only if all non-maximal jobs have already been scheduled, *i.e.* if $P = J' \setminus M$.

As all computations end up with Equation (4.34) due to the structure of recursive equations, the schedules counted by $N_X(\vec{B}, J', \varepsilon)$ necessarily cover $J' \setminus M$, or equivalently schedules which do not cover $J' \setminus M$ are withdrawn from the counting. Intuitively, the remaining work to be done by Inclusion-Exclusion is to ensure coverage of M, so we expect a sum with $2^{|M|}$ instead of $2^{|J'|}$ terms. This intuition agrees with the trimming technique of Nederlof [61] described in section 2.3.4: we remark that $N_X(\vec{B}, J', \varepsilon)$ is null whenever $X \subsetneq J' \setminus M$, because we always have $P \subset X$. So, the Inclusion-Exclusion formula to compute $N_X(\vec{B}, J', \varepsilon)$ reduces to:

$$N(\vec{B}, J', \varepsilon) = \sum_{J'\setminus M \subset X \subset J'} (-1)^{|J'|-|X|} N_X(\vec{B}, J', \varepsilon) = \sum_{\substack{Y\subset M \\ X=(J'\setminus M)\uplus Y}} (-1)^{|J'|-|X|} N_X(\vec{B}, J', \varepsilon) \tag{4.37}$$

Notice that the hybrid Inclusion-Exclusion procedure we use enables interleaving of maximal and non-maximal jobs, as shown on Figure 4.4 p. 104. Thus, it does not reduce to a dynamic programming scheme across subsets calling an auxiliary Inclusion-Exclusion computation, in which jobs managed by dynamic programming across subsets, *i.e.* non-maximal jobs, are scheduled before jobs managed by Inclusion-Exclusion, *i.e.* maximal jobs.



(a) hybrid Inclusion-Exclusion

(b) DP across subsets calling IE

Figure 4.4: Mixing Inclusion-Exclusion and Dynamic Programming across subsets

We derive the following complexity result.

Proposition 4.6. For an instance $\mathcal{I}$ of the $F|prmu, prec, r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem with $|M|$ maximal elements, the algorithm computes an optimal solution in $O^*(2^n||\mathcal{I}||^m)$ time and $O^*(2^{n-|M|}||\mathcal{I}||^m)$ space.

Proof. There are $O^*(2^{|J'\setminus M|}) = O^*(2^{n-|M|})$ downward closed sets of $J' \setminus M$. So, there are $O^*(2^{n-|M|}||\mathcal{I}||^m)$ dynamic programming states, and it takes $O^*(2^{n-|M|}||\mathcal{I}||^m)$ time and space to compute each $N_X(\vec{B}, J', \varepsilon)$. Besides there are $2^{|M|}$ terms in the reduced Inclusion-Exclusion sum corresponding to Equation (4.37) p. 103. □

Job precedences do not increase the time bound of the algorithm. But the space bound is no longer pseudopolynomial. Note that for some particular cases like *outtree*, where each job has at most one predecessor and precedences form a forest whose leaves are maximal jobs, $(n - |M|)$ can be substantially smaller than n. The symmetric situation, *intree*, can be managed by reverting jobs and reverting machines.

4.4.2 The case of chains

We now turn to the case where precedences are defined by a set of chains. Suppose there are c chains of length $\ell_1, \ldots, \ell_c$ (possibly 1 for trivial chains). The chains form a partition of the set of jobs: $J = \{j_{1,1} \prec \ldots \prec j_{1,\ell_1}, \quad \ldots, \quad j_{c,1} \prec \ldots \prec j_{c,\ell_c}\}$. The set of maximal elements is $M = \{j_{1,\ell_1} \ldots j_{c,\ell_c}\}$ and $|M| = c$. For $J' \subset J$, downward closed sets of $J' \setminus M$ are disjoint unions of beginning of chains, of the form $\{j_{1,1} \prec \ldots \prec j_{1,n_1}, \quad \ldots, \quad j_{c,1} \prec \ldots \prec j_{c,n_c}\}$ with $0 \leqslant n_1 < \ell_1, \ldots, 0 \leqslant n_c < \ell_c$. There are $\ell_1 \times \cdots \times \ell_c$ such sets, which implies the following result.

Proposition 4.7. For an instance $\mathcal{I}$ of the $F|prmu, chains, r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem, with c chains of length $\ell_1, \ldots, \ell_c$, the algorithm computes an optimal solution in $O^*(2^c \ell_1 \cdots \ell_c||\mathcal{I}||^m)$ time and $O^*(\ell_1 \cdots \ell_c||\mathcal{I}||^m)$ space.

Note that the result of Proposition 4.7 is not altered when considering, in addition to the chains, precedence constraints of the form $j' \prec j$ where j' is not the end of a chain and j is not the beginning of a chain: downward closed sets P are necessarily unions of beginning of chains, and the additional precedence constraints discard some of them. So, the expression $(\ell_1 \cdots \ell_c)$ remains an upper bound on the number of downward closed sets.

Proposition 4.7 p. 104 obviously reduces to the result of Proposition 4.3 p. 97 for n trivial chains of length 1. Knowing only the number c of chains, we derive the following result.

Proposition 4.8. For an instance $\mathcal{I}$ of the $F|prmu, chains, r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem with c chains, the algorithm computes an optimal solution in $O^*(2^c \sqrt[3]{3}^n ||\mathcal{I}||^m)$ time and $O^*(\sqrt[3]{3}^n ||\mathcal{I}||^m)$ space, with $\sqrt[3]{3} \simeq 1.4422$.

Proof. A product $\ell_1 \cdots \ell_c$ of non-negative integers knowing their sum $s = \ell_1 + \cdots + \ell_c$ is maximal when all integers (but possibly one) are equal to 3, and the product is bound by $O(3^{s/3})$. In our case, $s = n$. □

Minimizing a total cost

The study for the $F|prmu, prec, r_{ij}, \tilde{d}_{ij}|f_{\max}$ problem can be straightforwardly adapted to the $F|prmu, prec, r_{ij}, \tilde{d}_{ij}|\sum f_j$ problem. with time and space complexities multiplied by the optimum objective value γ^{opt}. We derive the following results.

Proposition 4.9. For an instance $\mathcal{I}$ of the $F|prmu, prec, r_{ij}, \tilde{d}_{ij}|\sum f_j$ problem with $|M|$ maximal elements, the algorithm computes an optimal solution in $O^*(2^n ||\mathcal{I}||^m \gamma^{opt})$ time and $O^*(2^{n-|M|} ||\mathcal{I}||^m \gamma^{opt})$ space.

Proposition 4.10. For an instance $\mathcal{I}$ of the $F|prmu, chains, r_{ij}, \tilde{d}_{ij}|\sum f_j$ problem, with c chains of length $\ell_1, \ldots, \ell_c$, the algorithm computes an optimal solution in $O^*(2^c \ell_1 \cdots \ell_c ||\mathcal{I}||^m \gamma^{opt})$ time and $O^*(\ell_1 \cdots \ell_c ||\mathcal{I}||^m \gamma^{opt})$ space.

Proposition 4.11. For an instance $\mathcal{I}$ of the $F|prmu, chains, r_{ij}, \tilde{d}_{ij}|\sum f_j$ problem with c chains, the algorithm computes an optimal solution in $O^*(2^c \sqrt[3]{3}^n ||\mathcal{I}||^m \gamma^{opt})$ time and $O^*(\sqrt[3]{3}^n ||\mathcal{I}||^m \gamma^{opt})$ space.

4.5 Towards non-permutation flowshop

In this section we focus on the $F||C_{\max}$ problem which consists in minimizing the makespan in a non-permutation flowshop.

In the $F|prmu|C_{\max}$ permutation flowshop problem, operations of different jobs cannot overtake each other. In other words, there is a common permutation π such that, for any fixed machine index i, operations $(O_{ij})_j$ are scheduled in chronological order $(O_{i,\pi(1)}, O_{i,\pi(2)}, \ldots, O_{i,\pi(n)})$. On the contrary, in the non-permutation flowshop problem $F||C_{\max}$, there are m potentially different permutations $\pi_1 \ldots \pi_m$ such that, for any fixed machine index i, operations $(O_{ij})_j$ are scheduled in chronological order $(O_{i,\pi_i(1)}, O_{i,\pi_i(2)}, \ldots, O_{i,\pi_i(n)})$.

The two problems $F|prmu|C_{\max}$ and $F||C_{\max}$ are not equivalent. Fig 4.5 p. 106 shows a minimal example, taken from Emmons and Vairaktarakis [22], for which the minimal makespan is smaller in the non-permutation flowshop than in the permutation flowshop. We consider an instance $\mathcal{I}$, common to both problems, with 4 machines but only 2 jobs. The

optimal makespan for this instance of the $F|prmu|C_{\max}$ problem equals 17. We describe the $2! = 2$ (semi-active) schedules, both optimal, corresponding to the permutations $\pi = (1, 2)$ and $\pi = (2, 1)$.

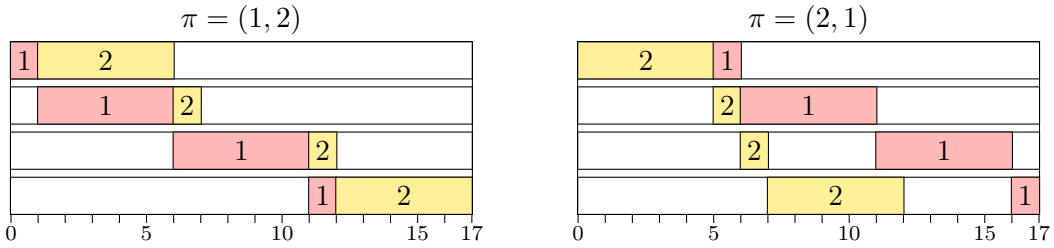


Figure 4.5: An instance of the $F4|prmu|C_{\max}$ problem and its two optimal solutions

The optimal makespan for the same instance $\mathcal{I}$ of the $F||C_{\max}$ problem equals 14. We can therefore improve the makespan by allowing several independent permutations, and the $F||C_{\max}$ problem does not reduce in general to the $F|prmu|C_{\max}$ problem. Figure 4.6 shows an optimal solution.

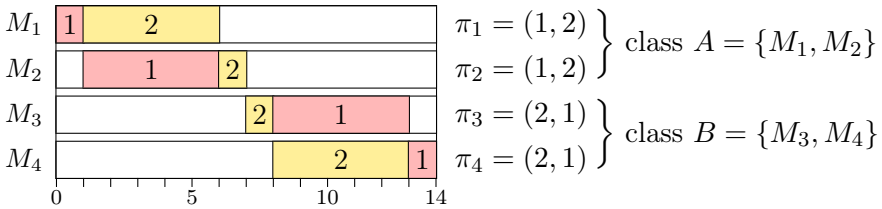


Figure 4.6: An instance of the $F4||C_{\max}$ problem and one of its optimal solutions

On the example of Figure 4.6 we can see that both first permutations are equal and that both last permutations are equal. It is not a coincidence. According to Emmons and Vairaktarakis [22], there is always an optimal schedule of the problem $F||C_{\max}$ whose first two permutations are equal and whose last two permutations are equal. Consequently, the $F2||C_{\max}$ problem reduces to the $F2|prmu|C_{\max}$ problem, and the $F3||C_{\max}$ problem reduces to the $F3|prmu|C_{\max}$ problem. As we have seen, this is not the case for a number of machines $m \geqslant 4$.

4.5.1 Inclusion-Exclusion formulation

We now describe an Inclusion-Exclusion formula suitable for non-permutation flowshop. For the sake of simplicity, we focus on the $F4||C_{\max}$ problem, but the formula can be adapted to other non-permutation flowshop problems.

As we have seen, in this problem we consider 4 machines but only 2 classes of identical permutations which we call A and B, containing machine numbers, as shown in Figure 4.6. We choose a representative from each class, for example the machine numbers $a = 1 \in A$ and $b = 3 \in B$. Normally, a (semi-active) schedule S is defined by its operations on all

machines, but we can reconstitute it from operations on machines a and b only, and finally it reduces to the list of operations $S = \binom{(O_{a,j_1},\ldots,O_{a,j_n})}{(O_{b,j'_1},\ldots,O_{b,j'_n})}$ and it is characterized by two permutations $\pi = (j_1,\ldots,j_n)$ and $\pi' = (j'_1,\ldots,j'_n)$.

We consider an admissibility criterion $\mathscr{A}(S)$ applied to schedules S. We now seek to calculate, by the Inclusion-Exclusion formula, the number N of (strict) admissible schedules. As in section 2.1.3, we define a set V of values to cover, a set V^* of relaxed schedules, and the set $\mathscr{R} = \{S \in V^* \mid \mathscr{A}(S)\}$ of admissible relaxed schedules. We derive N as the number of covering admissible relaxed schedules.

We actually cannot apply Inclusion-Exclusion to jobs, so instead of jobs we apply it to operations. From now on we identify the operations and their indices, *i.e.* we write (i,j) instead of O_{ij}. Denoting by $J = \{1,\ldots,n\}$ the set of all jobs and by $\uplus$ the union of two disjoint sets, we define the set of values to be covered as the set of operations, and we have:

$$V = \{(a,j), j \in J\} \uplus \{(b,j'), j' \in J\} = \{a\}\times J \ \uplus \ \{b\}\times J = \{a,b\} \times J \tag{4.38}$$

We take for V^* the set of lists made of a list of n operations on the machine a and a list of n operations on machine b, so we define:

$$V^* = (\{a\} \times J)^n \times (\{b\} \times J)^n \tag{4.39}$$

This amounts to imposing an additional admissibility criterion: operations must be processed by machines associated to a in the 1st list and to b in the 2nd list. We use the name Ω for subsets of operations, *i.e.* $\Omega \subset V$, and we derive from the Inclusion-Exclusion formula (2.16) p. 53:

$$N = \sum_{\Omega\subset V} (-1)^{|V|-|\Omega|} \operatorname{card}(\mathscr{R} \cap \Omega^*) = \sum_{\Omega\subset\{a,b\}\times J} (-1)^{2|J|-|\Omega|} \operatorname{card}(\mathscr{R} \cap \Omega^*) \tag{4.40}$$

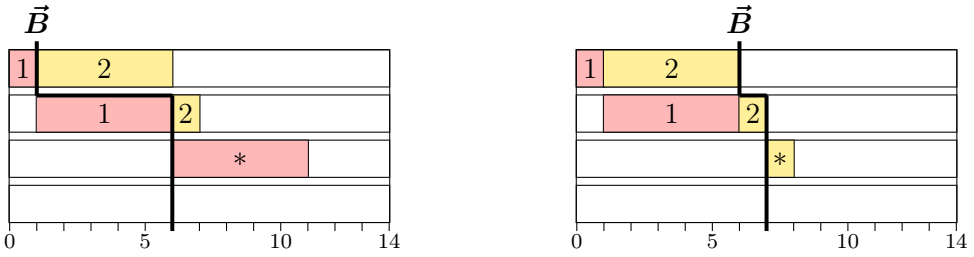
This formula can be written in a more symmetric and more explicit way by performing the change of variables $X = \{j \mid (a,j) \in \Omega\}$, $Y = \{j' \mid (b,j') \in \Omega\} \iff \Omega = \{a\}\times X \uplus \{b\}\times Y$:

$$N = \sum_{X\subset J} \sum_{Y\subset J} (-1)^{|J|-|X|}(-1)^{|J|-|Y|} \underbrace{\operatorname{card}\left(\mathscr{R} \cap (\{a\}\times X)^n \times (\{b\}\times Y)^n\right)}_{N_{XY}} \tag{4.41}$$

The expression N_{XY} counts the number of admissible relaxed schedules whose jobs of the 1st class of permutations are in X and whose jobs of the 2nd class of permutations are in Y.

4.5.2 Counting of relaxed schedules

To compute the quantity N_{XY} defined in Equation (4.41), the straightforward idea is to use a dynamic programming scheme involving release bound fronts $\vec{B}$. Unfortunately, the release bound front now depends not only on the previously scheduled operations, but also on the job of the next operation to be scheduled, as shown by Figure 4.7 p. 108.



(a) new operation ($*$) of job 1 (b) new operation ($*$) of job 2

Figure 4.7: Different release bound fronts when scheduling a new operation

So, a dynamic programming scheme has to integrate a release bound front per job, *i.e.* we must compute a number of the form $N_{XY}[\vec{B}_1, \ldots, \vec{B}_n]$ instead of $N_{XY}[\vec{B}]$. This leads to a space and time complexity to compute a single N_{XY} quantity in $O^*(||\mathcal{I}||^{mn})$ which is of no interest. We do not know any better scheme to compute N_{XY}.

4.6 Conclusions

In this chapter, we study exact algorithms to minimize objective functions defined as a maximum or total regular cost, for permutation flowshop scheduling problems with release dates and deadlines, *i.e.* the $F|prmu, r_{ij}, \tilde{d}_{ij}|f_{\max}$ and $F|prmu, r_{ij}, \tilde{d}_{ij}|\sum f_j$ problems. From a theoretical point of view, we focus on bounding worst-case time and space complexities.

The best general time and space complexity bound proved so far is due to Lenté et al. [50]. It is in $O^*(3^n)$ or in $O^*(2^n||\mathcal{I}||)$ time and space in case of 3-machine permutation flowshop to minimize makespan without release time nor deadline, *i.e.* for the $F3||C_{\max}$ problem. As for parallel machine scheduling, Jansen et al. [40] get a more general but less precise result: for the general $F|prmu, r_{ij}, \tilde{d}_{ij}|f_{\max}$ and $F|prmu, r_{ij}, \tilde{d}_{ij}|\sum f_j$ problems, they prove a time and space worst-case complexity in $O^*(c^n||\mathcal{I}||^{O(1)})$ for some constant c.

We describe an Inclusion-Exclusion based algorithm for the general $F|prmu, r_{ij}, \tilde{d}_{ij}|f_{\max}$ and $F|prmu, r_{ij}, \tilde{d}_{ij}|\sum f_j$ problems, using dynamic programming for enumerations. As for parallel machine scheduling, we take care to limit to a polynomial factor the overhead due to the calculations on the cardinals of sets involved in the Inclusion-Exclusion formula.

Finally, our main result is to describe an algorithm which, for any fixed number of machines m, runs with a moderately exponential time complexity, comparable to that of Jansen et al., yet more precise, and especially with an only pseudopolynomial space worst-case complexity bound. In addition, we extend our algorithm to the case of precedences between jobs. While the worst-case space complexity of our algorithm is no more polynomial, our algorithm saves an exponential amount of space compared to the state-of-the-art algorithm based on dynamic programming across subsets, without degrading the worst-case time complexity.

Table 4.8 summarizes the worst-case complexity bounds of our algorithm. We recall that the optimal objective value γ^{opt} is a pseudo-polynomial quantity, and we define μ as the number of jobs without a successor for the precedence partial order.

Table 4.8: Worst-case complexities for the permutation flowshop problem

problem	space	time
$F\|prmu, r_{ij}, \tilde{d}_{ij}\|f_{\max}$	$O^*(\|\|\mathcal{I}\|\|^m)$	$O^*(2^n\|\|\mathcal{I}\|\|^m)$
$F\|prmu, r_{ij}, \tilde{d}_{ij}\|\sum f_j$	$O^*(\|\|\mathcal{I}\|\|^m\gamma^{opt})$	$O^*(2^n\|\|\mathcal{I}\|\|^m\gamma^{opt})$
$F\|prmu, prec, r_{ij}, \tilde{d}_{ij}\|f_{\max}$	$O^*(2^{n-\mu}\|\|\mathcal{I}\|\|^m)$	$O^*(2^n\|\|\mathcal{I}\|\|^m)$
$F\|prmu, prec, r_{ij}, \tilde{d}_{ij}\|\sum f_j$	$O^*(2^{n-\mu}\|\|\mathcal{I}\|\|^m\gamma^{opt})$	$O^*(2^n\|\|\mathcal{I}\|\|^m\gamma^{opt})$

Contributions

- From a theoretical point of view, we demonstrate that a very general class of permutation flowshop problems with regular objectives can be solved with a moderately exponential time worst-case complexity and a pseudopolynomial space worst-case complexity.
- We extend these results to the case of precedences between jobs, and we derive a worst-case space complexity which is exponential but lower than the state-of-the-art complexity, without degrading the worst-case time complexity.
- These results were the subject of three presentations: one at the ROADEF2020 [65] french conference, another at the PMS2021 [66] international conference, and the specific result with precedences at the MAPSP2022 [68] international conference.
- These results are pubished in the Journal of Scheduling [71].

Chapter 5

Inclusion-Exclusion and Lagrangian Relaxation

Topics

- The great strength of Inclusion-Exclusion is to make it possible to work on schedules whose permutation constraint has been relaxed. In exchange, it needs to compute a very expensive, difficult to simplify, sum across subsets, including an exponential number of terms.
- To benefit from the advantages of Inclusion-Exclusion without bearing the inconveniences, we express Inclusion-Exclusion through Lagrangian penalties, instead of expressing it with a sum across subsets.
- We describe iterative methods to bound the optimal objective of a permutation scheduling problem based only on the solution of the relaxed problem.

As we saw in chapters 2 and 3, Inclusion-Exclusion is a valuable technique from a theoretical point of view, which solves a large class of scheduling problems with a moderate-exponential worst-case time complexity and a pseudopolynomial worst-case space complexity. But from a practical point of view, Inclusion-Exclusion turns out to be difficult to exploit in its direct form for the following reasons:

- It requires counting. Obviously, counting the number of schedules which satisfy a criterion is more difficult than simply knowing if there are any.
- The Inclusion-Exclusion formula requires the computation of a sum with an exponential number of terms. As we have seen in chapter 2, this sum is difficult to simplify, especially since it is an alternating sum.
- In general, counting is achieved by evaluating expressions involving sums, which are difficult to simplify, because all nonzero terms, even of low value, influence the result. Expressions involving minimums or maximums are easier to simplify, because many terms are dominated and do not affect the result. This is what makes the success of techniques like Branch-and-Bound.

The essence of Inclusion-Exclusion is to enable to relax the permutation constraint. This is its strength, because it is much easier to work on relaxed schedules than on strict ones. In particular, from the same mathematical equation we can derive a dynamic programming scheme on strict schedules and another one on relaxed schedules. But the dynamic programming scheme on strict schedules must include a job subset, which leads to an exponential number of states, while the same dynamic programming scheme on relaxed schedules has a pseudopolynomial number of states.

In its traditional form, the principle of Inclusion-Exclusion applied to schedules requires brutal enumeration of all possible job subsets. In this chapter, we propose to express the inclusion or exclusion of certain jobs through penalties instead of subsets: a negative penalty favors the inclusion of a job in a schedule, a positive penalty favors its exclusion.

We restrict ourselves to pure permutation problems, that is, to problems whose solution is entirely determined by the order of jobs. This is, in particular, the case of single-machine problems and permutation flowshop problems.

We recall the notations we use: given a job set $J = \{1, \dots, n\}$, a strict schedule is a schedule in which each job appears once and only once, so it is a permutation $S = (j_1, \dots, j_n) \in \mathfrak{S}_n$. On the contrary, a relaxed schedule is a schedule in which the jobs can be absent or duplicated, so it is an arbitrary list $S = (j_1, \dots, j_\ell) \in J^*$ where $J^* = \bigcup_{\ell \in \mathbb{N}} J^\ell$. Note that the length ℓ of a relaxed schedule is arbitrary and can be different from n.

Given a problem $P = \alpha|\beta|\gamma$, a schedule, strict or relaxed, is feasible when it meets all the constraints β of the problem. We denote by $\mathscr{R} \subset J^*$ the set of feasible relaxed schedules, and the set of feasible strict schedules is therefore $\mathscr{R} \cap \mathfrak{S}_n$. The problem $P = \alpha|\beta|\gamma$ consists in finding a strict feasible schedule which minimizes the objective γ.

5.1 Lagrangian relaxation of the permutation constraint

In a founding article, Abdul-Razaq and Potts [1] perform a Lagrangian relaxation of the permutation constraint of schedules. They are particularly interested in the $1|\textit{no-idle}|\sum f_j$ problem, but their method can be generalized to any permutation problem. We describe it in general terms, *i.e.* for a permutation problem $P = \alpha|\beta|\gamma$.

5.1.1 Principles

Each job j is assigned with a penalty $\lambda_j \in \mathbb{R}$. We denote by $\lambda = (\lambda_1, \dots, \lambda_n)$ the penalty vector. Penalties are added to the objective to be minimized, and we define the objective with penalties γ_λ by:

$$\forall\, S = (j_1, \dots, j_\ell), \quad \gamma_\lambda(S) = \gamma(S) + \sum_{j \in S} \lambda_j - \sum_{j=1}^{n} \lambda_j \tag{5.1}$$

Note that we subtract the term $\sum_{j=1}^{n} \lambda_j$. This term does not depend on the schedule S, but rather on λ. It will have no influence in the minimization of γ_λ. But for any strict schedule $S \in \mathfrak{S}_n$, the sums $\sum_{j \in S} \lambda_j$ and $\sum_{j=1}^{n} \lambda_j$ are equal and compensate each other. So, in this case, we get this very handy equality:

$$\forall\, S \in \mathfrak{S}_n, \quad \gamma_\lambda(S) = \gamma(S). \tag{5.2}$$

It is useful to define $n_j(S)$ as the number of occurrences of job j in schedule S. Formally:

$$\forall\, j \in J, \quad \forall\, S = (j_1, \ldots, j_\ell), \quad n_j(S) = \Big|\big\{k \in \{1, \ldots \ell\} \mid j_k = j\big\}\Big| \tag{5.3}$$

A schedule is strict if and only if each job appears once and only once:

$$\forall\, S \in J^*, \quad S \in \mathfrak{S}_n \iff \forall\, j \in J, n_j(S) = 1 \tag{5.4}$$

Equation (5.1) p. 112 can be rewritten as:

$$\forall\, S = (j_1, \ldots, j_\ell), \quad \gamma_\lambda(S) = \gamma(S) + \sum_{j=1}^{n} (n_j(S) - 1)\lambda_j \tag{5.5}$$

and for a strict schedule S, we always have $n_j(S) = 1$ which gives again $\gamma_\lambda(S) = \gamma(S)$.

Since we have introduced Lagrangian penalties, we are about to solve the Lagrangian dual problem resulting from the relaxation of the permutation constraint, and there appears a direct correspondence between Lagrangian problem and Inclusion-Exclusion, which we develop hereafter.

The primal problem from the Lagrangian point of view corresponds to the strict problem from the point of view of Inclusion-Exclusion, and consists of computing the strict optimum objective γ^{opt} and a corresponding strict optimal schedule S^{opt}. Formally:

$$\text{strict optimum objective:} \quad \gamma^{opt} = \min_{S \in \mathfrak{S}_n \cap \mathscr{R}} \gamma(S) \tag{5.6}$$

$$\text{strict optimal schedule:} \quad S^{opt} \in \mathfrak{S}_n \cap \mathscr{R} \;\mid\; \gamma(S^{opt}) = \gamma^{opt} \tag{5.7}$$

The Lagrangian dual problem consists in maximizing, when the penalties λ vary, the Lagrangian $\mathcal{L}(\lambda)$ which is the optimum objective of the primal problem in which the permutation constraints are relaxed and replaced by penalties. This latter problem corresponds, from the point of view of Inclusion-Exclusion, to the relaxed problem with penalties, which consists of calculating the optimum relaxed objective $\gamma_\lambda^{opt} = \mathcal{L}(\lambda)$ and a corresponding optimal relaxed schedule S_λ^{opt}. Formally:

$$\text{relaxed optimum objective:} \quad \gamma_\lambda^{opt} = \min_{S \in \mathscr{R}} \gamma_\lambda(S) \tag{5.8}$$

$$\text{relaxed optimal schedule:} \quad S_\lambda^{opt} \in \mathscr{R} \;\mid\; \gamma_\lambda(S^{opt}) = \gamma_\lambda^{opt} \tag{5.9}$$

We now assume that we can efficiently solve the relaxed primal problem: with fixed penalties λ, compute γ_λ^{opt} and S_λ^{opt}. Ideally, the aim is to solve the strict optimization problem, *i.e.* to calculate γ^{opt} and to obtain a strict optimal schedule S^{opt}. Failing this, the aim is to obtain a lower bound $LB \leqslant \gamma^{opt}$ as precise as possible, therefore as large as possible.

As γ and γ_λ coincide on $\mathfrak{S}_n$ and as $\mathscr{R} \cap \mathfrak{S}_n \subset \mathscr{R}$, we have $\forall\, \lambda, \gamma_\lambda^{opt} \leqslant \gamma^{opt}$. The idea is to approach the best possible lower bound derived from a λ, *i.e.* the Lagrangian dual optimum $\sup_\lambda \gamma_\lambda^{opt}$. Like all Lagrangian methods, this method provides a lower bound whose quality is limited by the "duality gap" $G = \gamma^{opt} - \sup_\lambda \gamma_\lambda^{opt}$.

5.1.2 Iterations

Abdul-Razaq and Potts approach the Lagrangian optimum $\sup_\lambda \gamma_\lambda^{opt}$ by a sub-gradient descent method. Algorithm 5.1 summarizes the method they propose. Some instructions, such as the computation of coefficient k or the stopping criterion, are left undefined and will be precisely described later on.

Algorithm 5.1: Computation of a lower bound of the optimum objective by a sub-gradient descent method

---

Procedure *SubGradientLB*:
  $\forall j,\ \lambda_j \leftarrow 0$
  $LB \leftarrow LB_0$ // *any lower bound*
  Initialize coefficient $k > 0$
  repeat
    $S \leftarrow S_\lambda^{opt}$ // *relaxed problem*
    $LB \leftarrow \max(LB, \gamma_\lambda^{opt})$
    if S *is strict* **then**
      Strict problem solved, S optimal. **Stop**.
    $\forall j,\ \lambda_j \leftarrow \lambda_j + k \cdot (n_j(S) - 1)$ (5.10)
    Update coefficient $k > 0$
  until *stop criterion*;
  Result: LB is a lower bound of γ^{opt}

Equation (5.10), which corresponds to the sub-gradient descent, is intuitive: for a job j absent from S, we have $(n_j(S) - 1) < 0$ so, at the next iteration the penalty λ_j is reduced, which favors j. On the contrary, for a job j duplicated in S, we have $(n_j(S) - 1) > 0$ so, λ_j is increased which penalizes j.

In the ideal case, the algorithm finds a strict optimal schedule, but in general the relaxed schedule S does not converge towards a strict schedule, and the main result of the algorithm is therefore LB, a lower bound of γ^{opt}. The LB variable can be initialized to any lower bound of γ^{opt}, for example 0 or any easy to compute lower bound, but it is in our interest to start from a lower bound which is close to the optimum.

The sub-gradient descent method is known for its convergence issues. If the coefficient k is too small, the method converges slowly. If k is too large, the method oscillates without converging. So, this method requires fine tuning, which has been the subject of many studies, in particular by van de Velde [88, p.27], whose formulas are directly used in Abdul-Razaq and Potts [1]. Here is a possible computation for k: we start by computing an upper bound UB of the strict optimum objective. It is a priori very easy, as any strict schedule $S \in \mathfrak{S}_n \cap \mathscr{R}$ provides an upper bound $UB = \gamma(S) \geqslant \gamma^{opt}$. It is still advantageous to get as close as possible of the strict optimum objective. Once the upper bound UB has been computed, we consider a variable $h > 0$ and we take:

$$k = h \frac{UB - \gamma_\lambda(S)}{\sum_{j=1}^{n} (n_j(S) - 1)^2} \tag{5.11}$$

We start from $h = 2$, and we divide h by 2 if the computed lower bound does not improve for a certain number of iterations (e.g. 5). Iterations end when h gets too small (for example after 10 divisions of h by 2, which corresponds to $h \leqslant 2^{-9} \simeq 2.10^{-3}$). This completes the description of Algorithm 5.1 p. 114.

5.2 Permutation classes

We introduce a new method to approach the Lagrangian optimum $\sup_\lambda \gamma_\lambda^{opt}$. Like the method of Abdul-Razaq and Potts, it performs an iterative computation of the penalties λ, and it only requires, at each step, to compute an optimal relaxed schedule for the corresponding penalties.

5.2.1 Principes

Consider the following relaxed schedule $S = (1, 2, 3, 3)$ which contains 4 jobs and in which the job 3 is duplicated. Now let us look at all the relaxed schedules which we can form by swapping the jobs of S. We obtain:

$$\begin{array}{llll} (1,2,3,3) & (1,3,2,3) & (1,3,3,2) & (2,1,3,3) \\ (2,3,1,3) & (2,3,3,1) & (3,1,2,3) & (3,1,3,2) \\ (3,2,1,3) & (3,2,3,1) & (3,3,1,2) & (3,3,2,1) \end{array} \tag{5.12}$$

All these relaxed schedules are equal to S up to a permutation and their set forms the permutation class of S. It turns out that the notion of permutation class is a notion of great importance.

In this example, schedules equal to S up to a permutation are also characterized by this property: these are the schedules containing exactly one job 1, one job 2, two jobs 3, and no other job.

Definition 5.1. Two relaxed schedules S and S' are equivalent, or equal up to a permutation, which we note $S \sim S'$, when each job j appears the same number of times in S and in S'. Formally:

$$\forall\, S, S' \in \mathscr{R},\quad S \sim S' \iff \forall\, j \in \{1, \ldots, n\}, n_j(S) = n_j(S') \tag{5.13}$$

The $\sim$ relation is clearly an equivalence relation, *i.e.* it is reflexive ($S \sim S$), symmetric ($S \sim S' \Leftrightarrow S' \sim S$) and transitive ($S \sim S' \sim S'' \Rightarrow S \sim S''$). So, we can define the equivalence classes:

$$\forall\, S \in \mathscr{R},\ \mathrm{Cl}(S) = \{S' \in \mathscr{R} \mid S \sim S'\}$$

As usual with classes, two schedules are equivalent when they share the same class: $S \sim S' \Leftrightarrow \mathrm{Cl}(S) = \mathrm{Cl}(S')$. Moreover, classes form a partition of the set $\mathscr{R}$ of feasible relaxed schedules: $\mathscr{R} = \bigcup_S \mathrm{Cl}(S)$ and $S \not\sim S' \Rightarrow \mathrm{Cl}(S) \cap \mathrm{Cl}(S') = \varnothing$. Any schedule $S \in \mathscr{R}$ belongs to its class ($S \in \mathrm{Cl}(S)$) and only to it. Note that strict schedules form a single class.

There are far fewer equivalence classes than relaxed schedules. Instead of eliminating non-optimal schedules individually, we have an interest in eliminating them class by class. For this, we will exploit the following proposition, very easy but very useful:

Proposition 5.2. The difference between objectives with penalties of two equivalent schedules does not depend on the penalties. Formally:

$$\forall\, S \sim S',\ \ \forall\, \lambda,\ \ \gamma_\lambda(S') - \gamma_\lambda(S) = \gamma(S') - \gamma(S) \text{ independently of } \lambda \tag{5.14}$$

Proof. Following Equation (5.5) p. 113, we have:

$$\gamma_\lambda(S') - \gamma_\lambda(S) = \gamma(S') - \gamma(S) + \sum_{j=1}^{n} \underbrace{(n_j(S') - n_j(S))}_{0} \lambda_j$$

We have $S \sim S'$, so $\forall\, j, n_j(S) = n_j(S')$ and each coefficient of λ_j in the sum is null. □

In order to concretely visualize this result, we represent on a diagram, in Figure 5.2, the objectives with penalties of the relaxed schedules, with classes on the horizontal axis and objectives on the vertical axis. Each dash represents a schedule. We assume that there are two classes: $\mathrm{Cl}(S) = \{S, S_1, S_2, S_3\}$ and $\mathrm{Cl}(S') = \{S', S'_1, S'_2\}$. The diagram on the left corresponds to a penalty vector λ. The diagram on the right corresponds to another penalty vector λ'.

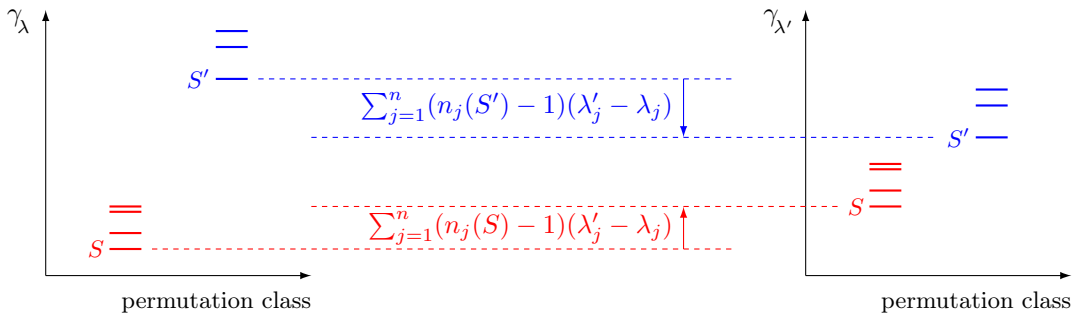


Figure 5.2: Evolution of two permutation classes when penalties change

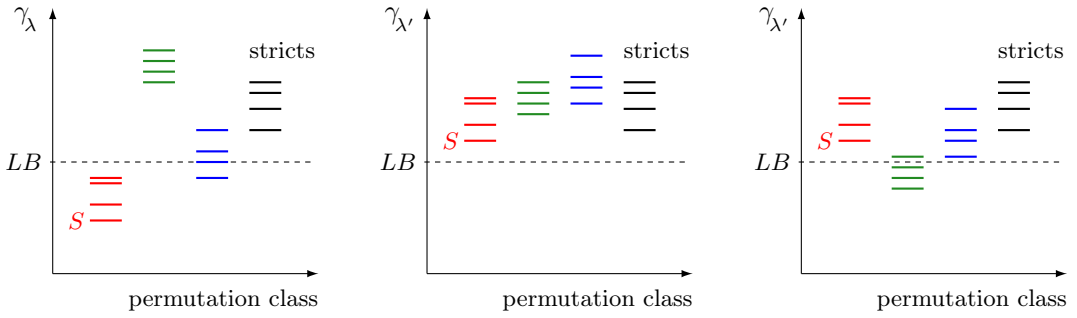
Here is the geometric translation of Proposition 5.2: when we change penalties from vector λ to λ', we shift the class $\mathrm{Cl}(S)$ by a certain height (precisely $\sum_{j=1}^{n}(n_j(S)-1)(\lambda'_j-\lambda_j)$) keeping its shape and the relative position of its schedules. Similarly, we shift the $\mathrm{Cl}(S')$ class by another height (precisely $\sum_{j=1}^{n}(n_j(S')-1)(\lambda'_j-\lambda_j)$) keeping its shape and the relative position of its schedules. Changing from λ to λ' modifies the relative position of classes but not the relative position of schedules within each class.

The following proposition will enable us to detect that entire classes are dominated, and to eliminate them of the search for the optimal schedule among the relaxed ones:

Proposition 5.3. Let $S \in \mathscr{R}$ be a relaxed schedule, λ and λ' penalty vectors. If S is optimal for λ but not for λ' then no $S' \sim S$ is optimal for λ', in other words the whole class $\mathrm{Cl}(S)$ is dominated for λ'.

Proof. Let $S' \sim S$. As S is optimal for λ, we have $\gamma_\lambda(S') \geqslant \gamma_\lambda(S)$ *i.e.* $\gamma_\lambda(S') - \gamma_\lambda(S) \geqslant 0$. By Proposition 5.2 p. 116, we have $\gamma_{\lambda'}(S') - \gamma_{\lambda'}(S) = \gamma_\lambda(S') - \gamma_\lambda(S) \geqslant 0$; moreover S is not optimal for λ', so $\gamma_{\lambda'}(S') \geqslant \gamma_{\lambda'}(S) > \gamma_{\lambda'}^{opt}$, therefore S' is not optimal for λ'. □

Our aim is to find a lower bound LB which approaches the Lagrangian dual optimum $\sup_\lambda \gamma_\lambda^{opt}$. Thanks to Proposition 5.3, we get a principle to refine our search. Suppose that a vector of penalties λ and a lower bound LB are already known. We partition the set $\mathscr{R}$ of feasible relaxed schedules in equivalence classes, and we represent schedules, as in Figure 5.3a, with classes on the horizontal axis and objectives on the vertical axis.



(a) Initially: S optimal for λ (b) Case S optimal for λ' (c) Case S not optimal for λ'

Figure 5.3: Refining a search for a dual Lagrangian optimum

We solve the relaxed problem for λ, so we find an optimal schedule $S = S_\lambda^{opt}$ for λ. Now, we assume that we can choose a new penalty vector λ' which improves the bound, *i.e.* such that $\gamma_{\lambda'}(S) > LB$. There are two cases: either S remains optimal for λ', or it is not anymore.

In the case where S remains optimal for λ' (Figure 5.3b), we have $\gamma_{\lambda'}^{opt} = \gamma_{\lambda'}(S) > LB$ so, we increase the bound by taking $LB \leftarrow \gamma_{\lambda'}(S)$, which improves it.

In the case where S is no longer optimal for λ' (Figure 5.3c), it is guaranteed by Proposition 5.3 that the whole class $\mathrm{Cl}(S)$ is dominated. This class is eliminated of the search, which simplifies the problem.

5.2.2 Algorithm

We derive an iterative method (Algorithm 5.4 p. 118) to compute a lower bound of the Lagrangian dual optimum. This algorithm is very close in structure to the sub-gradient descent (Algorithm 5.1 p. 114), but the way to update the penalties in statement (5.15) p. 118 is radically different.

Algorithm 5.4: Computation of a lower bound of the optimum objective by elimination of permutation classes

Procedure *PermutClassLB*:
  $\forall\, j,\ \lambda_j \leftarrow 0$
  $LB \leftarrow LB_0$ // *any lower bound*
  for $t \leftarrow 1, 2, 3, \ldots$ **do**
    $S_t \leftarrow S_\lambda^{opt}$ // *relaxed problem*
    $LB \leftarrow \max(LB, \gamma_\lambda^{opt})$
    if S *is strict* **then**
      Strict problem solved, S optimal. **Stop**.
    Find a new λ such that $\forall\, t' \leqslant t, \gamma_\lambda(S_{t'}) > LB$ (5.15)
    if *there is no such* λ **then**
      Stop
  Result: LB is a lower bound of γ^{opt}

Each constraint $\gamma_\lambda(S_{t'}) > LB$ appearing in instruction (5.15) is a linear inequality of continuous variables $\lambda_1, \ldots \lambda_n$. Indeed, according to Equation (5.5) p. 113, we have:

$$\gamma_\lambda(S_{t'}) > LB \iff \sum_{j=1}^{n} \underbrace{(n_j(S_{t'}) - 1)}_{\text{coefficient}} \lambda_j > \underbrace{LB - \gamma(S_{t'})}_{\text{right hand side}} \tag{5.16}$$

Since solving a system of strict inequalities makes little sense with continuous variables, we transform these strict inequalities into inclusive inequalities by choosing at each iteration t a value $\overline{LB}_t > LB_t$, where LB_t is the value of variable LB at iteration t. Obviously, the choice of $\overline{LB}_t$ influences the algorithm, and requires tuning: if $\overline{LB}_t$ is too large, there is a risk of not finding new penalties; if $\overline{LB}_t$ is too small and therefore too close to LB_t, the algorithm may converge very slowly. This tuning is presented in section 5.2.4.

It is convenient to introduce an additional variable M which represents the maximum of $\overline{LB}$ during iterations, *i.e.* $M = \max_{t' \leqslant t} \overline{LB}_{t'}$ at iteration t. This way, instruction (5.15) is translated into the following system with continuous variables $\lambda_1, \ldots, \lambda_n$ and M. Recall that S_t is the optimal relaxed schedule found at iteration t, and that $\gamma(S_t)$ is its strict objective.

$$\sum_{j=1}^{n} (n_j(S_1) - 1)\lambda_j - M \geqslant -\gamma(S_1) \tag{E_1}$$
$$\ldots$$
$$\sum_{j=1}^{n} (n_j(S_t) - 1)\lambda_j - M \geqslant -\gamma(S_t) \tag{E_t}$$
$$M \geqslant \overline{LB}_1 \tag{E'_1}$$
$$\ldots$$
$$M \geqslant \overline{LB}_t \tag{E'_t}$$

This system of linear inequalities is incremental: variables, *i.e.* $\lambda_1, \ldots, \lambda_n$ and M, are fixed once and for all and we do not add any during the iterations. At each iteration t, we add a constraint (E_t) on λ and M and a constraint (E'_t) on M, which reduce the polytope of solutions. If $\overline{LB}$ is chosen to be, like LB, increasing during the iterations, then the last inequality (E'_t) on M supersedes all other inequalities $(E'_1), \ldots, (E'_{t-1})$.

As it is possible to solve systems of continuous variable inequalities in polynomial time, and since we are not obliged to recalculate from scratch the solutions of this incremental system, we can hope to quickly perform each iteration in Algorithm 5.4 p. 118. A linear system has in general several solutions forming a polytope, therefore it remains to set the effective choice of a particular solution of this polytope. This setting will be presented in section 5.2.4.

5.2.3 A simple example

As an example, we run Algorithm 5.4 p. 118 on a small instance of the $1|\tilde{d}_j|\sum_j w_j C_j$ problem. We are not going into the details now, they will be specified in section 5.3 where this problem will be studied more deeply. This is a strongly NP-hard permutation problem. Obviously, the makespan of any strict schedule is independent on the order of jobs and equals $C_{\max} = \sum_{j=1}^{n} p_j$. As a special relaxation, we accept that a relaxed schedule does not contain exactly n jobs, but we impose that it has the same makespan as a strict schedule. So, a relaxed schedule S is feasible when it answers the deadlines and when $\sum_{j\in S} p_j = \sum_{j=1}^{n} p_j$.

We consider an instance with $n = 6$ jobs, described by Table 5.5. A systematic exploration of all feasible relaxed schedules shows that there are 3146 schedules, but they are divided into only 15 classes.

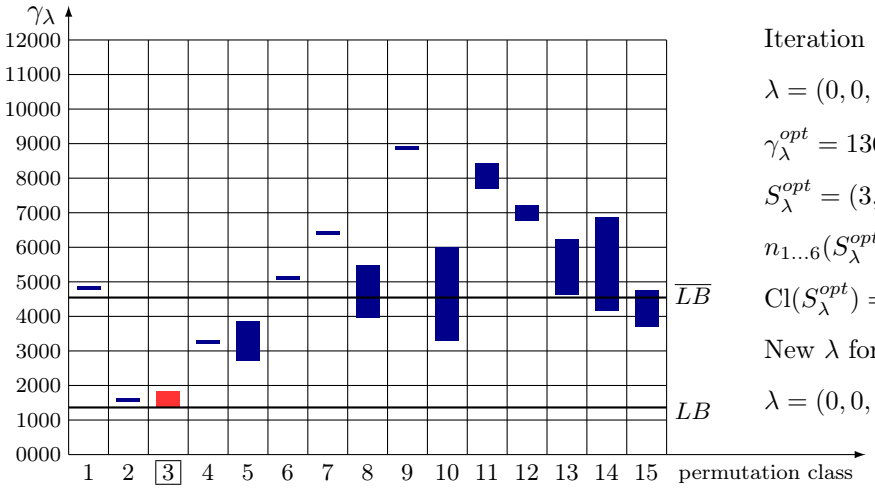
Table 5.5: Sample instance of the $1|\tilde{d}_j|\sum_j w_j C_j$ problem

j	p_j	w_j	d_j
1	47	5	416
2	74	7	252
3	32	1	413
4	90	2	236
5	84	1	376
6	53	6	400

We initially choose the upper bound UB as the objective of a strict feasible schedule, and we take at each iteration $\overline{LB} = UB + 1$. So, we have $LB \leqslant \gamma^{opt} < \overline{LB}$. We take a strict feasible ordering at random, e.g. $S = (6, 4, 2, 5, 1, 3)$, with objective $\gamma(S) = 4544$, yielding $\overline{LB} = 4545$. In a real situation we would use a better heuristic to obtain an upper bound close to the optimum objective.

Figure 5.6 p. 121 below describes in 5 tables the evolution of penalty vectors and objectives class by class. The class of strict schedules is represented on the right. For the sake of simplicity, we do not represent the individual schedules but the classes, by a rectangle describing the interval of the objectives of its schedules, *i.e.* class $\mathrm{Cl}(S)$ is represented by the

interval $[\min_{S' \sim S} \gamma_\lambda(S), \max_{S' \sim S} \gamma_\lambda(S)]$. Classes eliminated during successive iterations are shown in gray. Of course, this representation is artificial and is not actually computed by Algorithm 5.4 p. 118; it only illustrates its iterations.



Iteration # 1:

$\lambda = (0, 0, 0, 0, 0, 0)$

$\gamma_\lambda^{opt} = 1364$ = new LB

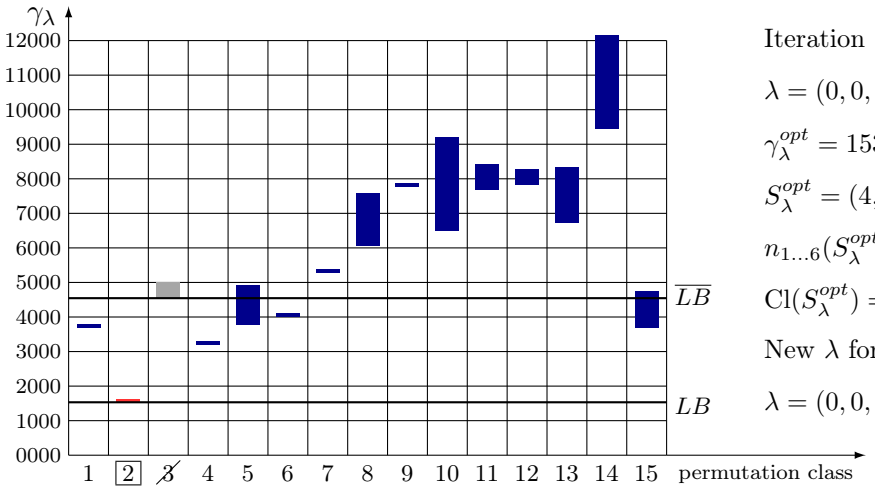
$S_\lambda^{opt} = (3, 3, 3, 5, 5, 5, 3)$

$n_{1\ldots6}(S_\lambda^{opt}) = (0, 0, 4, 0, 3, 0)$

$\mathrm{Cl}(S_\lambda^{opt})$ = class # 3

New λ for iteration # 2:

$\lambda = (0, 0, 1060.3, 0, 0, 0)$



Iteration # 2:

$\lambda = (0, 0, 1060.3, 0, 0, 0)$

$\gamma_\lambda^{opt} = 1532$ = new LB

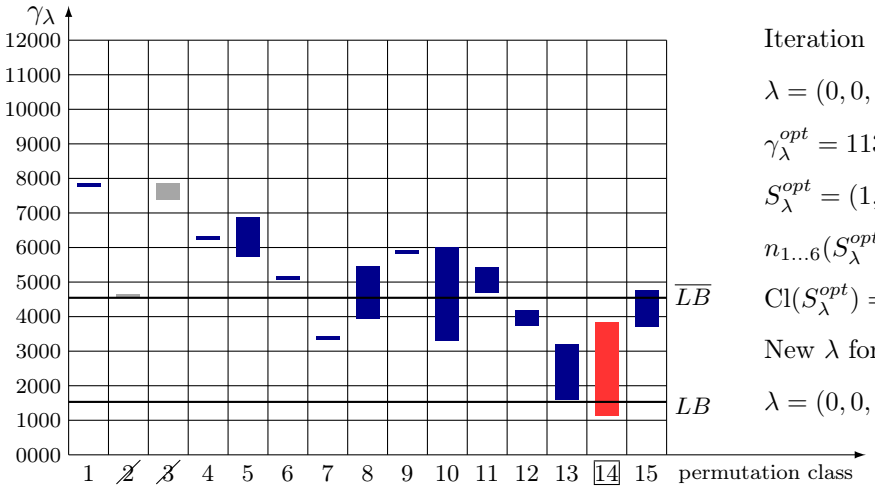
$S_\lambda^{opt} = (4, 4, 5, 5, 3)$

$n_{1\ldots6}(S_\lambda^{opt}) = (0, 0, 1, 2, 2, 0)$

$\mathrm{Cl}(S_\lambda^{opt})$ = class # 2

New λ for iteration # 3:

$\lambda = (0, 0, 0, 0, 3013.0, 0)$



Iteration # 3:

$\lambda = (0, 0, 0, 0, 3013.0, 0)$

$\gamma_\lambda^{opt} = 1137$, LB unchanged

$S_\lambda^{opt} = (1, 1, 1, 1, 3, 3, 3, 3, 3, 3)$

$n_{1\ldots6}(S_\lambda^{opt}) = (4, 0, 6, 0, 0, 0)$

$\mathrm{Cl}(S_\lambda^{opt})$ = class # 14

New λ for iteration # 4:

$\lambda = (0, 0, 681.6, 0, 3013.0, 0)$

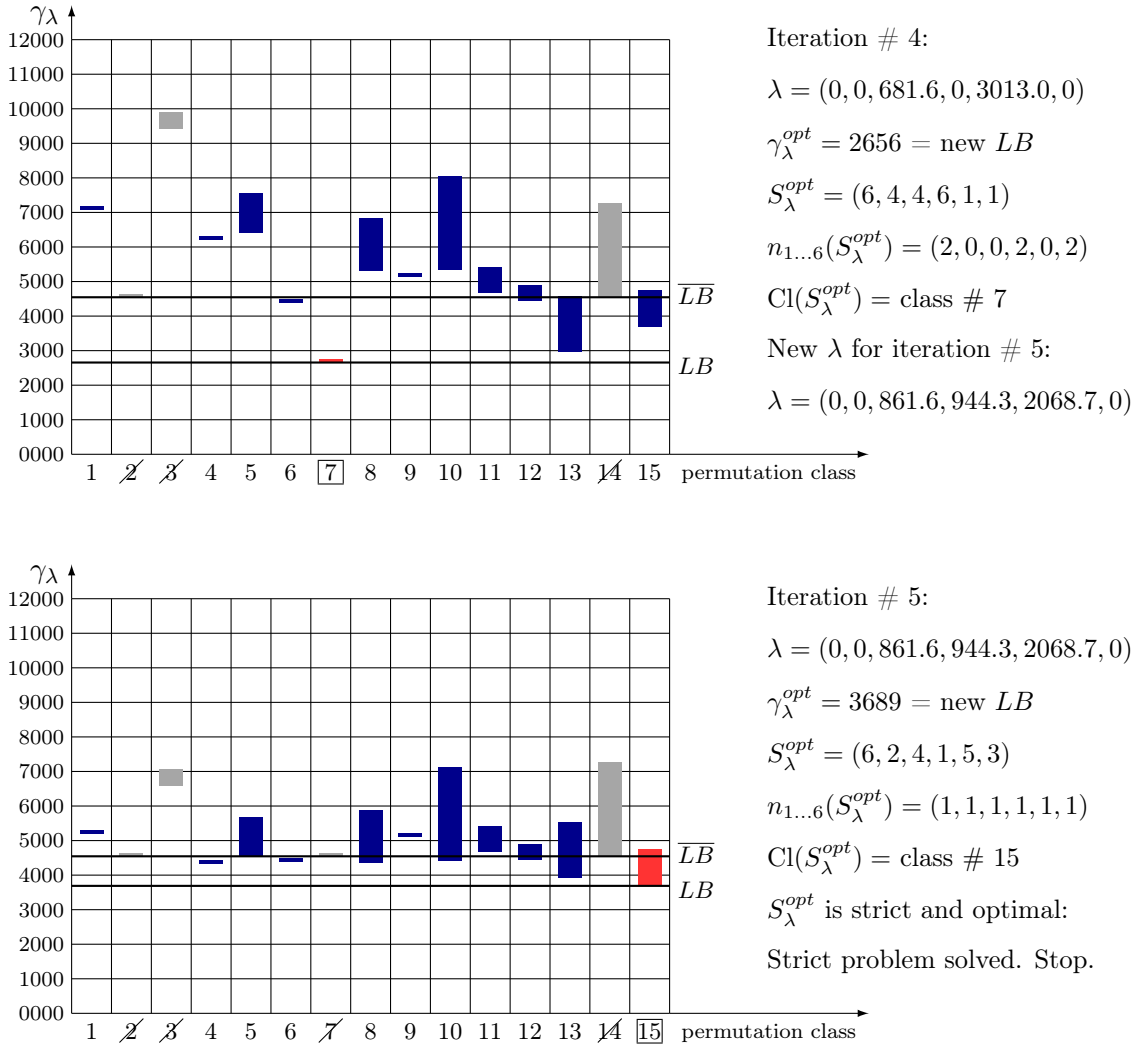


Figure 5.6: Iterations on permutations classes to derive a lower bound

As can be seen in figure 5.6, the objective of each class evolves iteration after iteration. The objective of the dominated classes continues to evolve, but above the $\overline{LB}$ limit. In our example, iteration # 3 does not improve the lower bound, but improves the internal state of the algorithm because it eliminates class # 14 by making it dominated. At iteration # 5, the optimal relaxed schedule is also strict, so the problem is solved to optimality. It is not always the case: if the duality gap is not null, this situation does not occur and iterations stop when the system of linear inequalities does not provide a new solution λ.

5.2.4 Penalty update policy

As we saw in section 5.2.2, the solution of strict inequalities (5.16) p. 118 used for updating λ penalties requires tweaking. Which solution λ shall we choose in the polytope of solutions of the system of inequalities? How much shall we increase the limit LB_t which determines the right hand sides of strict inequalities to transform them into inclusive inequalities? We hereafter provide answers to these questions.

Tuning of the linear inequations

We can simplify the presentation of the system ($E_1 \dots E_t$, $E'_1 \dots E'_t$) page 118: the set of inequalities ($E'_{t'}$) is equivalent to the single inequality $M \leqslant \max_{t' \in \{1,\dots,t\}} \overline{LB}_{t'}$. Since we assume that the bounds $\overline{LB}_t$ are increasing during the iterations, this inequality is reduced to $M \leqslant \overline{LB}_t$.

Also, variable M was introduced only to exhibit the incremental character of the system. Its value does not interest us, and there is always a solution where M is the largest possible, *i.e.* where $M = \overline{LB}_t$. From a theoretical point of view, we keep the inequality, but from a practical point of view it is possible to replace it with an equality. Finally, the system ($E_1 \dots E_t$, $E'_1 \dots E'_t$) can be translated as:

$$\left\{ \begin{array}{rcll} \sum_{j=1}^{n} (n_j(S_{t'}) - 1)\lambda_j - M & \geqslant & -\gamma(S_{t'}) & \forall\, t' \in \{1, \dots, t\} \\ M & \leqslant & \overline{LB}_t & \end{array} \right\} \tag{5.17}$$

In order to ensure a certain numerical stability, we want to keep the absolute values of λ_j as small as possible. With a linear system, it is possible to minimize the absolute value of a variable $v \in \mathbb{R}$: it suffices to write it in the form of two variables: $v = v^+ - v^-$ with $v^+ \geqslant 0$ and $v^- \geqslant 0$, and minimize the linear expression $v^+ + v^-$. Thus, we have $v^+ = \max(0, v)$, $v^- = \max(0, -v) = -\min(0, v)$ and $v^+ + v^- = |v|$. We apply this trick to each of the variables λ_j and we get this linear optimization system with $2n$ continuous variables $\lambda_j^+ \geqslant 0$, $\lambda_j^- \geqslant 0$, without counting $M \geqslant 0$:

$$\min \sum_{j=1}^{n} (\lambda_j^+ + \lambda_j^-) \ s.t. \ \left\{ \begin{array}{rcll} \sum_{j=1}^{n} (n_j(S_{t'}) - 1)(\lambda_j^+ - \lambda_j^-) - M & \geqslant & -\gamma(S_{t'}) & \forall\, t' \in \{1, \dots, t\} \\ M & \leqslant & \overline{LB}_t & \end{array} \right\} \tag{5.18}$$

In order to save variables, we can mutualize the negative part by considering a single λ^-, equal in absolute value to the largest negative part of a λ_j, which amounts to setting $\lambda^- = \max(0, \max_j -\lambda_j) = -\min(0, \min_j \lambda_j)$, and $\forall\, j, \lambda_j = \lambda_j^+ - \lambda^-$, so $\lambda_j^+ = \lambda_j + \lambda^- \geqslant 0$. We get this linear optimization system with $n+1$ continuous variables $\lambda_j^+ \geqslant 0$, $\lambda^- \geqslant 0$ without counting $M \geqslant 0$:

$$\min \lambda_j^- + \sum_{j=1}^{n} \lambda_j^+ \ s.t. \ \left\{ \begin{array}{rcll} \sum_{j=1}^{n} (n_j(S_{t'}) - 1)(\lambda_j^+ - \lambda^-) - M & \geqslant & -\gamma(S_{t'}) & \forall\, t' \in \{1, \dots, t\} \\ M & \leqslant & \overline{LB}_t & \end{array} \right\} \tag{5.19}$$

In practice, we will systematically use this system. We now have to study the computation of the bound $\overline{LB}_t > LB_t$ imposed, at iteration number t, on the linear system ($E_1 \dots E_t$, $E'_1 \dots E'_t$). We propose three policies: constant bound, constant increment, and adaptive increment.

Constant bound policy

We have a very great interest, if possible, in making that the optimal relaxed schedule for penalties λ is in fact strict, because in this case we solve the problem to optimality, which is much better than just deriving a lower bound on γ^{opt}.

In order to promote this possibility, it is in our interest to push the optimum of dominated relaxed classes beyond the γ^{opt} optimum of the strict class. We initially choose any upper bound UB of γ^{opt}, for example the objective of a strict schedule or any other value resulting of a heuristic. We deduce a strict bound $\overline{UB} > UB$. We take at each iteration $\overline{LB}_t = \overline{UB}$, we therefore have $LB_t \leqslant \gamma^{opt} < \overline{LB}_t$.

The main interest of the constant bound policy is to reduce the number of iterations. Without this policy, in Algorithm 5.4 p. 118, an optimal class $\mathrm{Cl}(S)$ for some penalties at some iteration may still be optimal at a following iteration: during the iterations, the objective $\gamma_\lambda(S)$ can only increase, but the objectives $\gamma_\lambda(S')$ of the schedules $S' \not\sim S$ of the others classes can increase even more and exceed $\gamma_\lambda(S)$.

On the contrary, with the constant bound policy, we impose $\overline{LB}_t > \gamma^{opt}$, so an optimal relaxed class at an iteration t is necessarily dominated by the strict class in following iterations, and can never again be the optimal of a relaxed problem, whatever the evolution of the penalties. Consequently, each iteration definitively eliminates at least one class and the number of iterations is bounded by the number of classes.

The constant bound policy offers guarantees on the number of iterations but it is a rather aggressive method in the sense that it imposes a bound $\overline{LB}_t$ potentially much higher than the dual Lagrangian optimum. We therefore have the risk of obtaining a system of unfeasible inequalities, to prematurely stop iterations and finally to obtain a bound LB of perfectible quality.

We now specify the choice of $\overline{UB}$. All values $\overline{UB} > \gamma^{opt}$ are suitable to prevent a non-strict class from being multiple times optimal. Taking a large value for the bound $\overline{UB}$ is unlikely to accelerate the convergence of LB, and strongly constrains the linear system, so risks to prematurely stop the iterations and to give in the end a poor lower bound LB. We therefore have an interest in choosing $\overline{UB}$ close to γ^{opt}, or equivalently close to UB, and we can set $\overline{UB} = UB + \delta$ by adding a value $\delta > 0$, small but numerically not negligible with respect to UB.

Constant increment policy

This simple policy consists in choosing an increment $\Delta LB > 0$, and systematically setting $\overline{LB}_t = LB_t + \Delta LB$. It can be improved in a simple way, by noticing, as for the constant bound policy, that it is useless to increase $\overline{LB}_t$ if it already exceeds γ^{opt}. So we initially choose a strict upper bound $\overline{UB} > \gamma^{opt}$, and we take:

$$\overline{LB}_t = \min(\overline{UB}, LB_t + \Delta LB) \tag{5.20}$$

The trouble with this policy is that the choice of increment ΔLB is tricky. If ΔLB is too large, we risk stopping the iterations too early and to miss the opportunity to improve the LB bound. There is also a risk of quickly deviating towards the constant bound policy, although this is not necessarily a disadvantage. According to Equation (5.20) p. 123, the constant increment policy meets the constant bound policy as soon as $LB \geqslant \overline{UB} - \Delta LB$, which can happen quickly if ΔLB is large.

If, on the contrary, the increment ΔLB is too small, we increase the chances to iterate and increase the bound LB, but we risk trampling: It may happen that the same relaxed schedule remains optimal for hundreds of iterations, during which the bound LB increases by ΔLB at each step. Such a situation leads the method to produce a bound LB of good quality, but very slowly.

Adaptive increment policy

In order to benefit from the best of the two policies that we have just studied, we propose an adaptive increment policy. The principle is as follows: we chain phases of constant increment policy. We start with a large enough increment so that the first phase reduces to a constant bound policy. Whenever the system of inequalities becomes unsolvable, we divide the increment by a quotient q, e.g. $q = 2$, and we go back to the system of inequalities. Iterations stop when ΔLB becomes too small, *i.e.* $\Delta LB < \Delta LB_{\min}$, where $\Delta LB_{\min}$ is a fixed parameter, e.g. $\Delta LB_{\min} = \frac{1}{q}$.

More precisely, we recall that at each iteration t we solve a system with t constraints on the λ_j's, determined by the optimal relaxed schedules already seen, and an additional constraint imposed by the bound $\overline{LB}_t$. If the system becomes unfeasible at iteration t, we replace $\overline{LB}_t = LB_t + \Delta\, LB$ by $\overline{LB}_t = LB_t + \frac{\Delta\, LB}{q} < LB_t + \Delta\, LB$, which has the effect of decreasing $\overline{LB}_t$, to relax the system and make it potentially feasible.

This relaxation is perfectly supported by the translation of the system in equation (5.19) p. 122. It is expensive because the solver must recalculate a new polytope of solutions instead of reducing it incrementally, but this is the only case where $\overline{LB}$ decreases. In all other cases it increases as initially expected in Algorithm 5.4 p. 118, and the system is incremental. Note that, under reasonable conditions, e.g. if the ratio $\frac{\Delta LB}{\gamma^{opt}}$ is initially bounded, the number of relaxations is bounded by $\log_q(\frac{\gamma^{opt}}{\Delta LB_{\min}})$, so it is low, and polynomial in the instance size at fixed q and $\Delta LB_{\min}$.

We extend Algorithm 5.4 p. 118 by integrating the adaptive increment policy, and we obtain Algorithm 5.7 p. 125, parameterized by the constants q and $\Delta LB_{\min}$, as well as by the initial values $LB_0 \leqslant \gamma^{opt}$, $\overline{UB}_0 > \gamma^{opt}$, and $\Delta LB_0 > 0$.

Algorithm 5.7: Computation of a lower bound using permutation classes and adaptive increment policy

---

Procedure *ExtendedPermutClassLB*:
- $\forall\, j,\ \lambda_j \leftarrow 0$
- $LB \leftarrow LB_0$ // *any lower bound of* γ^{opt}
- $\overline{UB} \leftarrow \overline{UB}_0$ // *any strict upper bound of* γ^{opt}
- $\Delta LB \leftarrow \Delta LB_0$ // *any non-negative value*
- **for** $t \leftarrow 1, 2, 3, \ldots$ **do**
  - $S_t \leftarrow S_\lambda^{opt}$ // *relaxed problem*
  - $LB \leftarrow \max(LB, \gamma_\lambda^{opt})$
  - **if** S *is strict* **then**
    - Strict problem solved, S optimal. **Stop**.
  - $\lambda \leftarrow$ optimum of system (5.19) p. 122 with $\overline{LB} = \min(\overline{UB}, LB + \Delta LB)$
  - **while** *there is no such* λ **do**
    - $\Delta LB \leftarrow \Delta LB / q$
    - **if** $\Delta LB < \Delta LB_{\min}$ **then**
      - **Stop**
    - // *the only case where linear system relaxation happens*
    - $\lambda \leftarrow$ optimum of system (5.19) p. 122 with $\overline{LB} = \min(\overline{UB}, LB + \Delta LB)$
- **Result:** LB is a lower bound of γ^{opt}

In the general case, Algorithm 5.7 extends Algorithm 5.4 p. 118 in that it does its bests to resume iterations when a system is unfeasible. Logically, and as we will see in section 5.3, it performs more iterations but it returns bounds of much better quality.

The constant bound policy and constant increment policy can be easily derived as particular cases of the adaptive increment policy. To emulate the constant bound policy, choose $\overline{UB}_0$ and take $LB_0 = 0$, $\Delta LB_0 = \overline{UB}_0$, $q = 2$ and $\Delta LB_{\min} = \Delta LB_0$ to prevent any linear system relaxation. The same way, to emulate the constant increment policy, choose ΔLB_0 and choose $\overline{UB}_0$, potentially $\overline{UB}_0 = +\infty$, and take $q = 2$ and $\Delta LB_{\min} = \Delta LB_0$. In the following, we will only use the adaptive increment policy.

5.3 Application to sequencing with no idle times

As a proof of concept, we study the application of permutation class based iterations (Agorithm 5.7) on some permutation scheduling problems, and we compare the computation time and the quality of the result compared to the iterative reference method from Abdul-Razaq and Potts (Agorithm 5.1 p. 114). These two methods are comparable because they produce the same type of result, *i.e.* a lower bound of the Lagrangian dual optimum, based on the same foundations, *i.e.* the solution of the same relaxed problems with penalties.

The main difficulty is to efficiently solve relaxed problems with penalties. Since penalties are a sum of individual costs, it is difficult to combine them with a maximum cost objective, and easy to combine them with a total cost objective.

In section 5.3.1, we study the generic solution of Abdul-Razaq and Potts, which applies to sequencing problems without idle times to minimize a total cost. Then, in section 5.3.2, we test it on the $1|\tilde{d}_j|\sum_j w_j C_j$ problem consisting in sequencing with deadline to minimize total weighted completion time. This problem is classic and well known, which makes it a good platform for experimentation. In particular, we can solve this problem to optimality, and therefore know, as an indication, the optimal objective, and evaluate the quality of the lower bounds of our algorithms.

5.3.1 Solution of the relaxed problem

In their article, Abdul-Razaq and Potts [1] propose a method which applies to the general $1|\textit{no-idle}|\sum_j f_j$ problem, with individual costs f_j not necessarily regular, and which we straightforwardly extend to the $1|\textit{no-idle}, \tilde{d}_j|\sum_j f_j$ problem. Note that, in the case where the individual costs f_j are regular, the $1|\tilde{d}_j|\sum_j f_j$ problem reduces to the $1|\textit{no-idle}, \tilde{d}_j|\sum_j f_j$ problem, because earliest schedules have no idle times and form a dominant set.

The $1|\textit{no-idle}, \tilde{d}_j|\sum_j f_j$ problem is a permutation problem with an interesting property: as idle times are prohibited, the completion time of a job only depends on the set of previously scheduled jobs, instead of depending on their sequence. In particular, the makespan of a schedule is necessarily the sum of the processing times of its jobs, regardless of their order.

The makespan of any strict schedule $S = (j_1, \ldots, j_n) \in \mathfrak{S}_n$ is $C_{\max} = \sum_{k=1}^{n} p_{j_k}$ which simplifies to $C_{\max} = \sum_{j=1}^{n} p_j$ regardless of S. Abdul-Razaq and Potts [1] exploit this fact, by relaxing the permutation constraint in a special way. The usual relaxation consists in accepting all the lists containing exactly n jobs and respecting the deadlines, which amounts to requiring, if we denote by J^* the set of job lists of any length:

$$\mathscr{R} = \left\{(j_1, \ldots, j_\ell) \in J^* \quad | \quad \forall\, k \leqslant \ell, \sum_{k' \leqslant k} p_{j_{k'}} \leqslant \tilde{d}_{j_k} \quad \wedge \quad \ell = n\right\} \tag{5.21}$$

Instead, the relaxation by Abdul-Razaq and Potts consists in accepting the lists of jobs whose makespan, or sum of processing times, is equal to the common makespan of any strict schedule, which amounts to requiring:

$$\mathscr{R} = \left\{(j_1, \ldots, j_\ell) \in J^* \quad | \quad \forall\, k \leqslant \ell, \sum_{k' \leqslant k} p_{j_{k'}} \leqslant \tilde{d}_{j_k} \quad \wedge \quad \sum_{k=1}^{\ell} p_{j_k} = \sum_{j=1}^{n} p_j\right\} \tag{5.22}$$

Notice that $\mathscr{R}$ contains all strict schedules: $\mathfrak{S}_n \subset \mathscr{R}$, and that all elements of $\mathscr{R}$ are feasible schedules.

We recall that the relaxed problem with penalties defined in section 5.1 consists, when penalties λ are fixed, in minimizing $\gamma_\lambda(S) = \sum_{j \in S} \lambda_j + f_j - \sum_{j=1}^{n} \lambda_j$ therefore to compute $\gamma_\lambda^{opt} = \min_{S \in \mathscr{R}} \gamma_\lambda(S)$. Abdul-Razaq and Potts define $opt[C]$ as the minimum objective of schedules of any length and completion time C. So, we have $\gamma_\lambda^{opt} = opt[\sum_{j=1}^{n} p_j]$, and $opt[C]$ is calculated by dynamic programming.

The only schedule with makespan $C = 0$ is the empty schedule, whose objective is the constant $\gamma_\lambda(()) = -\sum_{j=1}^{n} \lambda_j$. A non-empty schedule consists of a last job j, preceded by a prefix. The makespan C is greater than or equal to the processing time of j and less than

or equal to its deadline. The optimum is the minimum over all feasible j, it can be infinite if no job j is possible. We derive:

$$opt_\lambda[\,0\,] = -\textstyle\sum_{j=1}^{n} \lambda_j \tag{5.23}$$

$$opt_\lambda[C] = \min_{\substack{j\in\{1,\dots,n\} \\ p_j \leqslant C \leqslant \tilde{d}_j}} \Big(opt_\lambda[C - p_j] + f_j(C) + \lambda_j \Big) \qquad \text{for } C > 0 \tag{5.24}$$

The main advantage of this dynamic programming scheme is to be quite efficient: the number of states does not depend on the cost function and is bounded by the sum of the processing times. In practice, for benchmarks where the p_j's are randomly chosen from a uniform distribution between 1 and 100, the sum of n terms p_j is of the order of $50n$.

5.3.2 Computational experiments

We focus on the sequencing problem with deadlines to minimize total weighted completion time, denoted by $1|\tilde{d}_j|\sum_j w_j C_j$, and introduced in Example 1.3 p. 29. This problem is strongly NP-hard. As we have seen in section 1.3.3, the state-of-the-art exact algorithm to solve this problem is due to Shang et al. [81]. It can solve to optimality instances with up to approximately 130 jobs.

We evaluate iterative algorithms which compute lower bounds. For comparison, we take as a reference the state-of-the-art algorithm, *i.e.* the algorithm which computes, by analytical means, the best lower bound in very short time (say at most $O(n^2)$). To the best of our knowledge, it is the algorithm of Posner [73] improved by Bagchi and Ahmadi [3]. We denote this reference algorithm by (REF).

We now describe the algorithms we compare and the values we assign to the parameters they use. We denote by (ARP) the method of Abdul-Razaq and Potts (Algorithm 5.1 p. 114), which uses the lower bound LB_0, and the upper bound UB appearing in equation (5.11) p. 114. We denote by (IE) our method resulting from Inclusion-Exclusion, based on permutation classes, with adaptive increment policy (Algorithm 5.7 p. 125), which uses the lower bound LB_0, the strict upper bound $\overline{UB}_0$, the initial increment ΔLB_0, the increment divisor q and the minimum increment $\Delta LB_{\min}$.

To determine $UB \geqslant \gamma^{opt}$, we use Smith's heuristic: from the end to the beginning, among jobs which do not violate their deadline, schedule the one with the largest ratio p_j/w_j. This determines a schedule S, and we take $UB = \gamma(S)$. To determine the initial value LB_0 of the lower bound, we use the bound provided by the reference algorithm (REF). In a rather conservative way, we take as strict upper bound $\overline{UB}_0 = UB + 1$. We also take $\Delta LB_0 = \overline{UB}_0 - LB_0$, $q = 4$ and $\Delta LB_{\min} = \frac{1}{q} = \frac{1}{4}$.

Notice that we only present results of our (IE) method for the adaptive increment policy with these well-balanced parameters. The constant bound policy tends to produce low quality bounds, and the constant increment policy tends to be slow compared to the adaptive increment policy, without improving the bound quality. So, the corresponding results are not worth presenting.

We have implemented the three algorithms (REF), (ARP) and (IE) in C++, using CPLEX version 20.1 to solve systems of linear inequalities, on a 3Ghz Intel Core processor limited to a single thread. We tested 1200 randomly selected samples following a protocol described

by Abdul-Razaq and Potts [1] and taken over by Shang et al. [81], intended to produce instances with varying levels of difficulty. For each instance, we calculated, by the algorithm of Shang et al. [81], the exact value γ^{opt} of the optimum objective, and the lower bounds.

For each instance size n and each algorithm tested, we have computed the average CPU time, and the average deviation with respect to the optimum $dev = \frac{\gamma^{opt} - LB}{\gamma^{opt}}$. Table 5.8 summarizes the results we got. Notice that we do not indicate the memory consumption, because it is moderate for all the algorithms tested and it is not a bottleneck, so there is no memory issue.

Table 5.8: Evaluation of the Inclusion-Exclusion based lower bound

	CPU			deviation		
n	(ARP)	**(IE)**	(REF)	(ARP)	**(IE)**	(REF)
40	0.6s	**3.0s**	0.0s	0.58%	**0.29%**	0.47%
50	1.6s	**8.5s**	0.0s	0.55%	**0.21%**	0.33%
60	2.4s	**22.3s**	0.0s	0.60%	**0.19%**	0.28%
80	9.4s	**94.6s**	0.0s	0.63%	**0.14%**	0.20%

Experimentally, the computation time of our Inclusion-Exclusion based method (IE) seems to be in $\Theta(n^5)$, which is quite slow but polynomial. The (ARP) method is faster, but only 10 times faster, regardless of n. The deviation of the bound calculated by our method (IE) is excellent. It is not only much better than the bound calculated by the (ARP) method, but it is also systematically better than the reference method (REF).

5.4 Conclusions

Inspired by the concepts of Inclusion-Exclusion, and in particular the relaxation of the permutation constraint, we have developed a new iterative approach to derive a lower bound on the optimum objective of a permutation problem.

We conducted testings on the $1|\tilde{d}_j|\sum_j w_j C_j$ problem, *i.e.* sequencing with deadlines to minimize total completion times. These tests experimentally showed a computation time of $\Theta(n^5)$, which is high but polynomial, and moderate memory consumption. They also demonstrated that our new method based on Inclusion-Exclusion leads to a better approximation of the Lagrangian dual optimum than the sub-gradient method, and thus produces better quality bounds. Additionally, lower bounds produced by this approach beat in quality the state-of-the-art analytic bounds.

From a theoretical point of view, we applied our method to single-machine problems with no idle times. We can imagine applying it to other classes of problems, as two-machine flowshop problems, *i.e.* $F2||\sum_j f_j$. In this case, the main issue is to solve efficiently the relaxed problem. Indeed, the straightforward dynamic programming scheme is inefficient, because it needs to store completion times on both machines, which induces a number of states quadratic instead of linear in the sum of processing times.

Our tests were performed in vitro, *i.e.* standalone. There is still a lot of work to perform in vivo tests, *i.e.* to harmoniously integrate our technique in concrete programs. Such programs must exploit lower bounds and are therefore probably based on the Branch-and-Bound technique or its extensions. Since most lower bounds are based on analytical calculations and are therefore very fast, the main challenge is to identify when computing an Inclusion-Exclusion-based lower bound to cut a branch is worth the extra computation time.

Since the Inclusion-Exclusion based method constitutes an as-is replacement for the sub-gradient method, it seems possible to adapt a program exploiting the work of Abdul-Razaq and Potts. For example, their work has been extended and integrated by Tanaka et al. [83], who wrote a very efficient program to solve sequencing problems without idle-time, and which even constitutes the current state of the art in practice for the weighted tardiness minimization problem.

Thus, there are many perspectives for improving the Inclusion-Exclusion based technique of permutation classes and integrate it into programs to efficiently solve permutation scheduling problems.

Contributions

- In the context of permutation scheduling problems, we establish a link between Inclusion-Exclusion and Lagrangian penalties associated with jobs: both use the same relaxation of the initial problem.
- We develop a new iterative method to compute a lower bound of the optimum objective of a permutation problem, exploiting only the relaxed problem.
- Compared to sub-gradient methods, this method approaches the Lagrangian dual optimum more closely and therefore produces better quality lower bounds.
- These results were presented at the ROADEF2022 [70] conference

Conclusions

In this thesis, we have adapted the Inclusion-Exclusion technique to the field of scheduling. This technique consists of transforming he initial problem into a multitude of correlated relaxed problems and, most of the time, solving each relaxed problem by a pseudopolynomial dynamic programming scheme. By exploiting the correlations between relaxed problems, we contributed to a new technique, zero sweeping, enabling to solve all the relaxed problems together more efficiently than separately.

We have designed Inclusion-Exclusion based algorithms to solve a wide class of scheduling problems, with any unrelated parallel machine or permutation flowshop environment, with any release or deadline constraint, with any regular maximum or total cost objective. From a theoretical point of view, these algorithms achieve a moderate-exponential worst-case time complexity, along with a pseudopolynomial worst-case space complexity, and therefore enhance the state of the art of the literature.

More precisely, table 5.9 summarizes the worst-case complexities of our algorithms, in function of the number of machines m, the instance size n, the instance measure $||\mathcal{I}||$, and the optimum objective value γ^{opt}, itself polynomial in the instance measure. Thus, our algorithms achieve a worst-case space complexity in $O^*(||\mathcal{I}||^d)$ and a worst-case time complexity in $O^*(2^n||\mathcal{I}||^d)$, where the degree d depends on the problem and on the objective function.

Table 5.9: Worst-case complexities for parallel machine and permutation flowshop problems

problem	space	time
$R\|r_{ij}, \tilde{d}_{ij}\|f_{\max}$	$O^*(\|\|\mathcal{I}\|\|)$	$O^*(2^n\|\|\mathcal{I}\|\|)$
$R\|r_{ij}, \tilde{d}_{ij}\|\sum f_{ij}$	$O^*(\|\|\mathcal{I}\|\|\gamma^{opt})$	$O^*(2^n\|\|\mathcal{I}\|\|\gamma^{opt})$
$F\|prmu, r_{ij}, \tilde{d}_{ij}\|f_{\max}$	$O^*(\|\|\mathcal{I}\|\|^m)$	$O^*(2^n\|\|\mathcal{I}\|\|^m)$
$F\|prmu, r_{ij}, \tilde{d}_{ij}\|\sum f_j$	$O^*(\|\|\mathcal{I}\|\|^m\gamma^{opt})$	$O^*(2^n\|\|\mathcal{I}\|\|^m\gamma^{opt})$

We have studied another way of applying Inclusion-Exclusion. The strength of Inclusion-Exclusion is to replace a permutation problem, intrinsically difficult to solve, by a set of relaxed problems, much simpler to solve individually. We have made explicit the link between Inclusion-Exclusion and Lagrangian penalties, based on the same problem relaxation. We have introduced the notion of Inclusion-Exclusion based permutation classes, and we have contributed to a new iterative method to approximate the Lagrangian dual optimum, therefore deriving a lower bound on the optimal objective of a permutation problem.

Publications

This thesis has been an opportunity to submit publications to journals, and to expose presentations at conferences. Table 5.10 summarizes them. Titles have been adapted and translated.

Table 5.10: Publications submitted and conferences attended during this thesis

reference	title
ROADEF2020 [65]	An exponential, Inclusion-Exclusion based algorithm for permutation flowshop to minimize makespan
JOSH [71]	Moderate worst-case complexity bounds for the permutation flowshop scheduling problem using Inclusion-Exclusion
ROADEF2021 [67]	Inclusion-Exclusion based solution of parallel machine scheduling problems
PMS2021 [66]	An Inclusion-Exclusion based algorithm for the permutation flowshop scheduling problem with regular objective
JOCO [72]	Exponential-time algorithms for parallel machine scheduling problems
ROADEF2022 [70]	Inclusion-Exclusion based Lagrangian iterations to solve permutation scheduling problems
PMS2022 [69]	An Inclusion-Exclusion based general exponential-time algorithm for the solution of unrelated parallel machine scheduling problems
MAPSP2022 [68]	An Inclusion-Exclusion based algorithm for permutation flowshop scheduling with job precedences

Perspectives

This work leads us to develop some perspectives, both from theoretical and practical point of views, both short term and long term. We develop them below.

From a theoretical point of view, chapters 3 and 4 show that parallel machines as well as permutation flowshop problems can be solved with a worst-case time complexity in $O^*(2^n||\mathcal{I}||^d)$ and a worst-case space complexity in $O^*(||\mathcal{I}||^d)$, where the degree d depends on the problem and the objective, and is derived from table 5.9. Is it possible to combine both problems together? In this case, we would solve the hybrid flowshop problem, where each job is processed in m successive stages $i = 1, \ldots, m$ in this order, and where each

stage i comprises m_i parallel machines. The first difficulty is to combine both problems in a unified dynamic programming scheme. The second difficulty is to keep the optimization of chapter 3 with parallel machines, *i.e.* to obtain a pseudopolynomial part of the complexity in $O^*(||\mathcal{I}||)$ instead of $O^*(||\mathcal{I}||^{m_i})$.

Is it possible to extend the results of chapters 3 and 4 to multi-criteria objectives? Multi-criteria scheduling was popularized by T'kindt and Billaut [84]. It consists in minimizing several criteria at the same time. The difficulty is that optimization does not lead to a single minimum objective value, but to several tuples of minimal criteria, called the Pareto optima. Taking these Pareto optima into account makes countings harder.

In chapter 4, we took into account precedences between jobs, at the cost of a no longer pseudopolynomial worst-case space complexity, but gaining an exponential factor on the worst-case space complexity and without degrading the worst-case time complexity. Is it possible to adapt the management of precedences to parallel machine problems? The main difficulty is that precedences only influence the order of the jobs in a permutation flowshop problem, whereas they also influence the assignment of machines to jobs in a parallel machine problem.

In chapters 3 and 4, we gave upper bounds on worst-case time and space complexities for parallel machine or permutation flowshop problems. It would be interesting to give some lower bounds on these complexities. Admitting the Exponential Time Hypothesis of Impagliazzo and Paturi [38], Jansen et al. [40] showed in general that many scheduling problems can not be solved in sub-exponential time, *i.e.* in $o(c^n)$ with $c > 1$. Shang [78] independently got the same result in the particular case of the three machine flowshop problem $F3||C_{\max}$. Is it possible to get more accurate results, especially to explicit a constant c? It is a question of fine-grained complexity, and some articles, e.g. Cygan et al. [19], are investigating similar issues, but much work remains to be done before getting a response.

Every time we used Inclusion-Exclusion to solve a scheduling problem, we were brought to apply it to sets of jobs, or possibly tasks, and to count the relaxed schedules by a dynamic programming scheme involving time fronts. This way of proceeding seems the most natural, but are there other ways (for example on sets of machines or time intervals), and if so are they effective?

Moreover, we only used dynamic programming to count relaxed schedules, because countings naturally induce recurrence formulas involving sums, which themselves are expressed naturally by dynamic programming. As mentioned in section 2.2, we can replace dynamic programming by power calculations on adjacency matrices in the case of unweighted directed graphs. It seems possible to extend this technique to scheduling problems, considering, in the usual way, the (weighted) scheduling graph whose nodes are jobs or tasks and whose arcs are weighted by the processing times, and turning it into an unweighted graph by adding dummy nodes. Is it possible to get benefits from this technique?

There are other radically different counting techniques. For example, Burnside's and Pólya's theorems allow to perform fine countings in the case of groups operating on sets (see e.g. Beekman [7]). It seems difficult to apply this technique to scheduling problems, because, in order to get a group structure, it is necessary to identify symmetries, which are rare for scheduling problems. But is it really impossible?

From a more practical point of view, we studied in chapter 5 an iterative algorithm to derive a lower bound on the optimum objective value of a permutation problem. For now, we have focused on the $1|\tilde{d}_j|\sum w_jC_j$ problem. We need to study the speed of convergence and the quality of the lower bounds obtained for other sequencing problems, as the $1||\sum w_jT_j$ or $1|r_j, \textit{no-idle}|\sum f_j$ problems. We can also study permutation flowshop problems, such as the $F2||\sum C_j$ problem.

For iterative methods, based on Inclusion-Exclusion or on subgradient descent, to be efficient, we must be able to quickly solve relaxed problems with penalties. We have an efficient method using dynamic programming for problems of type $1|\textit{no-idle}|\sum f_j$, but dynamic programming is inefficient in the case of flowshop problems. Moreover, objectives of type $f_{\max}$ are incompatible with dynamic programming, because a problem of type $\alpha|\beta|f_{\max}$ results in a relaxed problem with penalties of type $\alpha|\beta|(\max_j f_j + \sum_j \lambda_j)$, which is not nicely handled by dynamic programming. Are there efficient alternative methods to solve relaxed problems with penalties, such as constraint programming or mixed integer linear programming?

In the case where the iteration based method we presented in chapter 5 produces bounds of better quality than the analytical lower bounds, the most promising way to use it is to integrate it into a Branch-and-Bound algorithm. The challenge is to use this method only when the improvement of the lower bound compensates the slowness of the computation. In a Branch-and-Bound algorithm, we know in advance a threshold lower bound ε not to be exceeded, and we save time by cutting a node of the branching tree when we can demonstrate that the lower bound LB at this node meets the condition $LB > \varepsilon$. Knowing ε in advance, we can hope to demonstrate that $LB > \varepsilon$ without fully computing LB, thus saving many iterations.

Finally, our iterative method based on Inclusion-Exclusion is a direct replacement for the subgradient method, so we can adapt to Inclusion-Exclusion any algorithm exploiting a lower bound based on the sub-gradient. The current state of the art in practice for the $1||\sum w_jT_j$ problem is due to Tanaka et al. [83], and it explicitly exploits the sub-gradient based method. It therefore seems promising to adapt this algorithm to our Inclusion-Exclusion based lower bound.

Here are now some longer-term perspectives for continuing this work: from a theoretical point of view, we have to develop better mathematical tools for complexity analysis. Currently, we are often at a loss to analyze the complexity of algorithms. Our upper bounds are often very coarse, and it may happen to improve the worst-case complexity bounds of well-known algorithms by improving their analysis, but without modifying them. This is for example what Björklund et al. [9] did for the traveling salesman problem with a graph of bounded degree.

Moreover, the worst-case analysis, though very interesting from a theoretical point of view, is often too disconnected from the real efficiency, and therefore from the practical interest of an algorithm. Inclusion-Exclusion is an example of this: an Inclusion-Exclusion based algorithm is often close to the worst case scenario, whereas a Branch-and-Bound based algorithm is often extremely far from it and more effective in practice, without anyone knowing how to prove it.

The average case analysis, the one that best reflects the real efficiency, relies on the expectation of time and space required to process an instance. It more difficult and less well mastered than the worst-case analysis, and moreover it relies on the probability of each instance to occur. It is therefore necessary to integrate these probabilities from the very beginning of the modeling process of a problem.

From a practical point of view, we must put more efforts on concretely improving scheduling algorithms and allow them to solve instances of larger sizes. We are only at the beginning: a lot of problems are currently solved to optimality only for instances of very small sizes.

By comparison, huge progress has been made in Operations Research on problems previously known for their intractability: we now solve instances of the traveling salesman problem with several tens of thousands of nodes, and instances of the boolean satisfiability problem with several tens of thousands of variables, whereas these achievements seemed forever out of reach a few decades ago.

There is hope in the long term: on some problems, enormous progress has been made. For example, Carlier's algorithm [14] enables to solve instances with several thousand jobs for the sequencing problem $1|r_j, \tilde{d}_j|$- with interval constraints. Similarly, the $F3||C_{\max}$ three-machine flowshop problem to minimize makespan is solved more and more effectively, and the current state of the art, due to Gmys et al. [30], solves instances with several thousands of tasks.

This quest for improvement also involves identifying compromises to accelerate the solution of problems while keeping them close to reality. For example, considering planar graphs accelerated the solution of the traveling salesman problem in gigantic proportions. So there is a lot to be gained by developing models which result in less difficult problems while remaining realistic.

This thesis is a link in a long scientific chain to bring answers to some of these questions. The thesis of Shang [78], entitled "Exact Algorithms With Worst-case Guarantee For Scheduling: From Theory to Practice", is the previous link in this chain. I am eager to see the next one.

Bibliography

[1] T. S. Abdul-Razaq and C. Potts. Dynamic Programming State-Space Relaxation for Single-Machine Scheduling. *Journal of the Operational Research Society*, 39(2): 141–152, 1988. doi:10.2307/2582377. [pp. 42, 112, 114, 126, 128]

[2] S. Ashour. A Branch-and-Bound Algorithm for Flow Shop Scheduling Problems. *American Institute of Industrial Engineers Transactions*, 2(2):172–176, 1970. doi:10.1080/05695557008974749. [p. 42]

[3] U. Bagchi and R. H. Ahmadi. An Improved Lower Bound for Minimizing Weighted Completion Times with Deadlines. *Operations Research*, 35(2):311–313, 1987. doi:10.1287/opre.35.2.311. [pp. 42, 127]

[4] S. P. Bansal. Single machine scheduling to minimize weighted sum of completion times with secondary criterion — A branch and bound approach. *European Journal of Operational Research*, 5(3):177–181, 1980. doi:10.1016/0377-2217(80)90087-9. [p. 42]

[5] E. T. Bax. Inclusion and Exclusion Algorithm for the Hamiltonian Path Problem. *Information Processing Letters*, 47(4):203–207, 1993. doi:10.1016/0020-0190(93)90033-6. [pp. 16, 55, 56]

[6] E. T. Bax. Recurrence-Based Heuristics for the Hamiltonian Path Inclusion and Exclusion Algorithm. Technical report, California Institute of Technology, USA, 1994. URL `https://resolver.caltech.edu/CaltechCSTR:1994.cs-tr-94-11`. [pp. 45, 55, 56, 64]

[7] R. M. Beekman. *An Introduction to Number Theoretic Combinatorics*. Lulu.com, 2017. ISBN 978-1-3299-9116-3. [p. 133]

[8] A. Bjorklund and T. Husfeldt. Inclusion–Exclusion Algorithms for Counting Set Partitions. In *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 575–582, nov 2006. doi:10.1109/FOCS.2006.41. [pp. 16, 45]

[9] A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. The travelling salesman problem in bounded degree graphs. In *International Colloquium on Automata, Languages, and Programming*, pages 198–209. Springer, 2008. doi:10.1145/2151171.2151181. [p. 134]

[10] S. A. Brah and J. L. Hunsucker. Branch and Bound algorithm for the flow shop with multiple processors. *European Journal of Operational Research*, 51(1):88–99, 1991. doi:10.1016/0377-2217(91)90148-O. [p. 42]

[11] A. P. G. Brown and Z. A. Lomnicki. Some Applications of the "Branch-and-Bound" Algorithm to the Machine Scheduling Problem. *Journal of the Operational Research Society*, 17(2):173–186, 1966. doi:10.1057/jors.1966.25. [p. 42]

[12] P. Brucker. *Scheduling Algorithms*. Springer, 5th edition, 2010. ISBN 978-3-6420-8907-7. [pp. 19, 37]

[13] R. N. Burns. Scheduling to minimize the weighted sum of completion times with secondary criteria. *Naval Research Logistics Quarterly*, 23(1):125–129, 1976. doi:10.1002/nav.3800230111. [p. 42]

[14] J. Carlier. The one-machine sequencing problem. *European Journal of Operational Research*, 11(1):42–47, 1982. doi:10.1016/S0377-2217(82)80007-6. [pp. 28, 42, 135]

[15] J. Carlier and I. Rebaï. Two branch and bound algorithms for the permutation flow shop problem. *European Journal of Operational Research*, 90(2):238–251, 1996. doi:10.1016/0377-2217(95)00352-5. [p. 42]

[16] J. Cheng, H. Kise, and H. Matsumoto. A branch-and-bound algorithm with fuzzy inference for a permutation flowshop scheduling problem. *European Journal of Operational Research*, 96(3):578–590, 1997. doi:10.1016/S0377-2217(96)00083-5. [p. 42]

[17] S. A. Cook. The Complexity of Theorem-Proving Procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. Association for Computing Machinery. doi:10.1145/800157.805047. [p. 15]

[18] M. Cygan, M. Pilipczuk, M. Pilipczuk, and J. O. Wojtaszczyk. Scheduling Partially Ordered Jobs Faster Than 2^n. In *Algorithms – ESA 2011*, pages 299–310. Springer, 2011. doi:10.1007/978-3-642-23719-5_26. [pp. 39, 40]

[19] M. Cygan, H. Dell, D. Lokshtanov, D. Marx, J. Nederlof, Y. Okamoto, R. Paturi, S. Saurabh, and M. Wahlström. On Problems as Hard as CNF-SAT. *ACM Transactions on Algorithms*, 12(3):1–24, 2016. doi:10.1145/2925416. [p. 133]

[20] F. Della Croce, V. T'kindt, and O. Ploton. Parallel machine scheduling with minimum number of tardy jobs: Approximation and exponential algorithms. *Applied Mathematics and Computation*, 397, 2021. doi:10.1016/j.amc.2020.125888. [p. 29]

[21] K. Dohmen. *Improved Bonferroni Inequalities via Abstract Tubes: Inequalities and Identities of Inclusion-Exclusion Type*. Lecture Notes in Mathematics. Springer, 2003. ISBN 978-3-5403-9399-3. [pp. 64, 66]

[22] H. Emmons and G. Vairaktarakis. *Flow Shop Scheduling: Theoretical Results, Algorithms, and Applications*. International Series in Operations Research & Management Science. Springer, 2013. ISBN 978-1-4614-5151-8. doi:10.1007/978-1-4614-5152-5. [pp. 105, 106]

[23] F.V. Fomin and D. Kratsch. *Exact Exponential Algorithms*. Springer, 2010. ISBN 978-3-6421-6532-0. [pp. 16, 39, 40, 41, 49, 53, 64, 74]

[24] J. M. Framinan, J. N. D. Gupta, and R. Leisten. A review and classification of heuristics for permutation flow-shop scheduling with makespan objective. *Journal of the Operational Research Society*, 55(12):1243–1255, 2004. doi:10.1057/palgrave.jors.2601784. [p. 92]

[25] J. Galambos and I. Simonelli. *Bonferroni-type inequalities with applications*. Springer, 1996. ISBN 978-0-3879-4776-1. [p. 60]

[26] H.L. Gantt. *Work, Wages and Profits*. Hive Publishing Company, 1910. ISBN 978-0-8796-0048-8. [pp. 17, 20]

[27] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, first edition edition, 1979. ISBN 978-0-7167-1045-5. [p. 31]

[28] M.R. Garey, D.S. Johnson, and R. Sethi. The complexity of flowshop and jobshop scheduling. *Mathematics of Operations Research*, 1(2):117–129, 1976. doi:10.1287/moor.1.2.117. [p. 92]

[29] M. Garraffa, L. Shang, F. Della Croce, and V. T'kindt. An exact exponential branch-and-merge algorithm for the single machine total tardiness problem. *Theoretical Computer Science*, 745:133–149, 2018. doi:10.1016/j.tcs.2018.05.040. [p. 41]

[30] J. Gmys, M. Mezmaz, N. Melab, and D. Tuyttens. A computationally efficient Branch-and-Bound algorithm for the permutation flow-shop scheduling problem. *European Journal of Operational Research*, 284(3):814–833, 2020. doi:10.1016/j.ejor.2020.01.039. [pp. 42, 135]

[31] X. Goaoc, J. Matoušek, P. Paták, Z. Safernová, and M. Tancer. Simplifying Inclusion-Exclusion formulas. In *European Conference on Combinatorics, Graph Theory and Applications*, Pisa, Italy, 2013. URL `https://hal.inria.fr/hal-00764182`. [p. 63]

[32] R. L. 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*, 5(2):287–326, 1979. doi:10.1016/S0167-5060(08)70356-X. [pp. 17, 21]

[33] R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison-Wesley Longman Publishing Co., 2nd edition, 1994. ISBN 978-0-2015-5802-9. [p. 59]

[34] D. Harvey and J. Van Der Hoeven. Integer multiplication in time $O(n \log n)$. *Annals of Mathematics*, 193(2):563–617, 2020. doi:10.4007/annals.2021.193.2.4. [p. 85]

[35] M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial and Applied mathematics*, 10(1):196–210, 1962. doi:10.1137/0110015. [p. 56]

[36] E. Horowitz and S. Sahni. Computing Partitions with Applications to the Knapsack Problem. *Journal of the ACM*, 21(2):277–292, 1974. doi:10.1145/321812.321823. [p. 40]

[37] E. Ignall and L. Schrage. Application of the Branch and Bound Technique to Some Flow-Shop Scheduling Problems. *Operations Research*, 13(3):400–412, 1965. doi:10.1287/opre.13.3.400. [p. 42]

[38] R. Impagliazzo and R. Paturi. Complexity of k-SAT. In *Proceedings. Fourteenth Annual IEEE Conference on Computational Complexity (Formerly: Structure in Complexity Theory Conference)*, pages 237–240, 1999. doi:10.1109/CCC.1999.766282. [pp. 15, 74, 92, 133]

[39] J. R. Jackson. Scheduling a production line to minimize maximum tardiness. Technical Report 43, University of California, Los Angeles, 1955. [pp. 26, 28]

[40] K. Jansen, F. Land, and K. Land. Bounding the Running Time of Algorithms for Scheduling and Packing Problems. In *Algorithms and Data Structures - 13th International Symposium*, volume 8037, pages 439–450. Springer, 2013. doi:10.1007/978-3-642-40104-6_38. [pp. 71, 74, 76, 88, 92, 108, 133]

[41] S. M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1(1):61–68, 1954. doi:10.1002/nav.3800010110. [pp. 30, 92]

[42] R. Karp. Dynamic programming meets the principle of inclusion and exclusion. *Operations Research Letters*, 1(2):49–51, 1982. doi:10.1016/0167-6377(82)90044-X. [pp. 16, 43, 45, 56, 58, 74, 75, 79, 82, 89]

[43] S. Kohn, A. Gottlieb, and M. Kohn. A Generating Function Approach to the Traveling Salesman Problem. In *Proceedings of the 1977 Annual Conference*, pages 294–300, 1977. doi:10.1145/800179.810218. [pp. 16, 45, 56]

[44] M. Koivisto. An $O^*(2^n)$ Algorithm for Graph Coloring and Other Partitioning Problems via Inclusion–Exclusion. In *Proceedings - Annual IEEE Symposium on Foundations of Computer Science, FOCS*, pages 583–590, 2006. doi:10.1109/FOCS.2006.11. [pp. 16, 45]

[45] T. Ladhari and M. Haouari. A computational study of the permutation flow shop problem based on a tight lower bound. *Computers and Operations Research*, 32(7): 1831–1847, 2005. doi:10.1016/j.cor.2003.12.001. [p. 42]

[46] B. J. Lageweg, J. K. Lenstra, and A. H. G. Rinnooy Kan. A General Bounding Scheme for the Permutation Flow-Shop Problem. *Operations Research*, 26(1):53–67, 1978. doi:10.1287/opre.26.1.53. [p. 42]

[47] E.L. Lawler. Optimal sequencing of a single machine subject to precedence constraints. *Management Science*, 19(5):544–546, 1973. doi:doi.org/10.1287/mnsc.19.5.544. [p. 29]

[48] J.K. Lenstra, A.H.G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977. doi:10.1016/S0167-5060(08)70743-X. [p. 92]

[49] C. Lenté, M. Liedloff, A. Soukhal, and V. T'Kindt. On an extension of the Sort and Search method with application to scheduling theory. *Theoretical Computer Science*, 511:13–22, 2013. doi:10.1016/j.tcs.2013.05.023. [p. 40]

[50] C. Lenté, M. Liedloff, A. Soukhal, and V. T'Kindt. Exponential Algorithms for Scheduling Problems. Technical report, University of Tours, 2014. URL `https://hal.archives-ouvertes.fr/hal-00944382`. [pp. 39, 40, 74, 88, 108]

[51] L. A. Levin. Universal Sequential Search Problems. *Problems of Information Transmission*, 9(3):115–116, 1973. [p. 15]

[52] Z. A. Lomnicki. A "Branch-and-Bound" Algorithm for the Exact Solution of the Three-Machine Scheduling Problem. *Journal of the Operational Research Society*, 16 (1):89–100, 1965. doi:10.1057/jors.1965.7. [p. 42]

[53] D. R. Mazur. *Combinatorics: a Guided Tour*. Mathematical Association of America, Washington, DC, 2010. ISBN 978-0-8838-5762-5. [pp. 50, 63]

[54] G. B. McMahon and P. G. Burton. Flow-Shop Scheduling with the Branch-and-Bound Method. *Operations Research*, 15(3):473–481, 1967. doi:10.1287/opre.15.3.473. [p. 42]

[55] J. M. Moore. An n Job, One Machine Sequencing Algorithm for Minimizing the Number of Late Jobs. *Management Science*, 15(1):102–109, 1968. doi:10.1287/mnsc.15.1.102. [p. 29]

[56] D. Naiman and H. Wynn. Inclusion-Exclusion-Bonferroni Identities and Inequalities for Discrete Tube-Like Problems via Euler Characteristics. *The Annals of Statistics*, 20(1):43–76, 1992. doi:10.1214/aos/1176348512. [pp. 64, 65]

[57] H. Narushima. Principle of inclusion-exclusion on semilattices. *Journal of Combinatorial Theory, Series A*, 17(2):196–203, 1974. doi:10.1016/0097-3165(74)90006-5. [p. 64]

[58] H. Narushima. Principle of inclusion-exclusion on partially ordered sets. *Discrete Mathematics*, 42(2):243–250, 1982. doi:10.1016/0012-365X(82)90221-7. [p. 64]

[59] J. Nederlof. Inclusion exclusion for hard problems. Master's thesis, Utrecht University, 2008. [pp. 16, 45, 53, 58]

[60] J. Nederlof. Fast polynomial-space algorithms using möbius inversion: Improving on steiner tree and related problems. In *Proceedings of the 36th International Colloquium on Automata, Languages and Programming: Part I*, ICALP '09, pages 713–725. Springer, 2009. doi:10.1007/978-3-642-02927-1_59. [pp. 16, 63]

[61] J. Nederlof. Fast Polynomial-Space Algorithms Using Inclusion-Exclusion. *Algorithmica*, 65:868–884, 2013. doi:10.1007/s00453-012-9630-x. [pp. 16, 45, 66, 71, 103]

[62] J. Nederlof, J. M. M. van Rooij, and Th. C. van Dijk. Inclusion/exclusion meets measure and conquer. *Algorithmica*, 69(3):685–740, 2014. doi:10.1007/s00453-013-9759-2. [pp. 16, 64]

[63] Y. Pan. An improved branch and bound algorithm for single machine scheduling with deadlines to minimize total weighted completion time. *Operations Research Letters*, 31(6):492–496, 2003. doi:10.1016/S0167-6377(03)00048-8. [p. 42]

[64] M. L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. Springer, 5th edition, 2016. ISBN 978-3-319-26580-3. [pp. 19, 27, 37]

[65] O. Ploton and V. T'kindt. Un algorithme exponentiel basé sur Inclusion-Exclusion pour la résolution d'un problème d'ordonnancement de type flowshop. In *21ème congrès de la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF'20)*, Montpellier, France, 2020. [pp. 109, 132]

[66] O. Ploton and V. T'kindt. An Inclusion-Exclusion based algorithm for the permutation flowshop scheduling problem. In *17th International Workshop on Project Management and Scheduling (PMS'20/21)*, Toulouse, France, online, 2021. [pp. 109, 132]

[67] O. Ploton and V. T'kindt. Résolution de problèmes d'ordonnancement sur machines parallèles par Inclusion-Exclusion. In *22ème congrès de la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF'21)*, Mulhouse, France, online, 2021. [pp. 89, 132]

[68] O. Ploton and V. T'kindt. An Inclusion-Exclusion based algorithm for permutation flowshop scheduling with job precedences. In *15th Workshop on Models and Algorithms for Planning and Scheduling (MAPSP'22)*, Oropa Biella, Italy, 2022. [pp. 109, 132]

[69] O. Ploton and V. T'kindt. An Inclusion-Exclusion based general exponential-time algorithm for the solution of unrelated parallel machine scheduling problems. In *18th International Workshop on Project Management and Scheduling (PMS'22)*, Ghent, Belgium, online, 2022. [pp. 89, 132]

[70] O. Ploton and V. T'kindt. Étude de l'intégration de techniques d'algorithmique exponentielle pour la résolution d'un problème d'ordonnancement par une méthode arborescente. In *23ème congrès de la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF'22)*, Lyon, France, 2022. [pp. 129, 132]

[71] O. Ploton and V. T'kindt. Moderate worst-case complexity bounds for the permutation flowshop scheduling problem using Inclusion-Exclusion. *Journal of Scheduling*, 2022. doi:10.1007/s10951-022-00759-1. [pp. 109, 132]

[72] O. Ploton and V. T'kindt. Exponential-time algorithms for parallel machine scheduling problems. *Journal of Combinatorial Optimization*, 44:3405–3418, 2022. doi:10.1007/s10878-022-00901-x. [pp. 89, 132]

[73] M. E. Posner. Minimizing Weighted Completion Times with Deadlines. *Operations Research*, 33(3):562–574, 1985. doi:10.1287/opre.33.3.562. [pp. 42, 127]

[74] C. N. Potts. An adaptive branching rule for the permutation flow-shop problem. *European Journal of Operational Research*, 5(1):19–25, 1980. doi:10.1016/0377-2217(80)90069-7. [p. 42]

[75] C. N. Potts and L. N. Van Wassenhove. An algorithm for single machine sequencing with deadlines to minimize total weighted completion time. *European Journal of Operational Research*, 12(4):379–387, 1983. doi:10.1016/0377-2217(83)90159-5. [p. 42]

[76] M. Ritt. A branch-and-bound algorithm with cyclic best-first search for the permutation flow shop scheduling problem. In *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 872–877, 2016. doi:10.1109/COASE.2016.7743493. [p. 42]

[77] A. Schönhage and V. Strassen. Schnelle Multiplikation großer Zahlen. *Computing*, 7 (3):281–292, 1971. doi:10.1007/BF02242355. [p. 85]

[78] L. Shang. *Exact Algorithms With Worst-case Guarantee For Scheduling: From Theory to Practice*. PhD thesis, Université de Tours, nov 2017. [pp. 92, 133, 135]

[79] L. Shang and V. T'Kindt. A Sort and Search method for multicriteria optimization problems with applications to scheduling theory. *Journal of Multi-Criteria Decision Analysis*, 26(1-2):84–90, 2019. doi:10.1002/mcda.1660. [p. 41]

[80] L. Shang, C. Lenté, M. Liedloff, and V. T'Kindt. Exact exponential algorithms for 3-machine flowshop scheduling problems. *Journal of Scheduling*, 21(2):227–233, 2018. doi:10.1007/s10951-017-0524-2. [pp. 39, 40, 92, 96]

[81] L. Shang, V. T'Kindt, and F. Della Croce. Branch and Memorize exact algorithms for sequencing problems: Efficient embedding of memorization into search trees. *Computers and Operations Research*, 128:105171, 2021. doi:10.1016/j.cor.2020.105171. [pp. 29, 42, 127, 128]

[82] W. E. Smith. Various optimizers for single-stage production. *Naval Research Logistics Quarterly*, 3(1-2):59–66, 1956. doi:10.1002/nav.3800030106. [p. 29]

[83] S. Tanaka, S. Fujikuma, and M. Araki. An exact algorithm for single-machine scheduling without machine idle time. *Journal of Scheduling*, 12(6):575–593, 2009. doi:10.1007/s10951-008-0093-5. [pp. 129, 134]

[84] V. T'kindt and J. C. Billaut. *Multicriteria Scheduling: Theory, Models and Algorithms*. Springer, 2006. ISBN 978-3-540-28230-3. [p. 133]

[85] V. T'kindt, F. Della Croce, and C. Esswein. Revisiting Branch and Bound Search Strategies for Machine Scheduling Problems. *Journal of Scheduling*, 7:429–440, 2004. doi:10.1023/B:JOSH.0000046075.73409.7d. [p. 42]

[86] V. T'kindt, L. Shang, and F. Della Croce. Exponential time algorithms for just-in-time scheduling problems with common due date and symmetric weights. *Journal of Combinatorial Optimization*, 39:764–775, 2020. doi:10.1007/s10878-019-00512-z. [pp. 40, 41]

[87] C. P. Tomazella and M. S. Nagano. A comprehensive review of Branch-and-Bound algorithms: Guidelines and directions for further research on the flow-shop scheduling problem. *Expert Systems with Applications*, 158:113556, 2020. doi:10.1016/j.eswa.2020.113556. [p. 41]

[88] S. L. van de Velde. *Machine scheduling and Lagrangian relaxation*. PhD thesis, Mathematics and Computer Science, 1991. [p. 114]

[89] D. P. Williamson, L. A. Hall, J. A. Hoogeveen, C. A. J. Hurkens, J. K. Lenstra, S. V. Sevast'janov, and D. B. Shmoys. Short Shop Schedules. *Operations Research*, 45(2): 288–294, 1997. doi:10.1287/opre.45.2.288. [p. 92]

[90] G. J. Woeginger. Exact Algorithms for NP-Hard Problems: A Survey. In *Combinatorial Optimization — Eureka, You Shrink!: Papers Dedicated to Jack Edmonds 5th International Workshop Aussois, France, March 5–9, 2001 Revised Papers*, pages 185–207. Springer, 2003. doi:10.1007/3-540-36478-1_17. [p. 33]

Olivier PLOTON

Apports de l'Inclusion-Exclusion à la résolution exacte ou approchée de problèmes d'ordonnancement

Résumé :

Nous étudions la résolution de problèmes d'ordonnancement par Inclusion-Exclusion. Cette formule de combinatoire permet d'évaluer le nombre de solutions de problèmes de couverture ou de permutation, et par contrecoup d'en expliciter des solutions optimales.
D'un point de vue théorique, nous résolvons à l'optimalité et avec une complexité au pire cas modérément exponentielle en temps et pseudopolynomiale en espace, tout problème d'ordonnancement à machines parallèles ou d'atelier à cheminement unique, avec n'importe quelle contrainte temporelle d'intervalle et n'importe quel objectif régulier du type coût maximum ou coût total.
La force de l'Inclusion-Exclusion est de simplifier des problèmes en relâchant leur contrainte de couverture. Nous établissons un lien entre Inclusion-Exclusion et pénalités Lagrangiennes, et nous décrivons une nouvelle méthode itérative pour minorer l'objectif optimum d'un problème de permutation, fondée uniquement sur la résolution de problèmes relâchés.

Mots clefs: ordonnancement, complexité au pire cas, méthodes exactes, approximation, Inclusion-Exclusion.

Abstract:

We study the resolution of scheduling problems by Inclusion-Exclusion. This combinatorial formula makes it possible to evaluate the number of solutions of coverage or permutation problems, and consequently to explicit optimal solutions.
From a theoretical point of view, we solve to optimality and with a moderately exponential worst-case time complexity and a pseudopolynomial worst-case space complexity, any parallel machine or permutation flowshop scheduling problem, with any interval time constraint and any maximum cost or total cost regular objective.
The strength of Inclusion-Exclusion is to simplify problems by relaxing their coverage constraint. We establish a link between Inclusion-Exclusion and Lagrangian penalties, and we describe a new iterative method to derive a lower bound of the optimum objective of a permutation problem, based only on the resolution of relaxed problems.

Keywords: scheduling, worst case complexity, exact methods, approximation, Inclusion-Exclusion.