UNIVERSITÉ FRANÇOIS RABELAIS DE TOURS

ÉCOLE DOCTORALE MIPTIS

Laboratoire d'Informatique (EA 6300)

THÈSE présentée par :

Lei SHANG

soutenue le : 30/11/2017

pour obtenir le grade de : Docteur de l'Université François-Rabelais de Tours

Discipline/ Spécialité : INFORMATIQUE

Exact Algorithms With Worst-case Guarantee For Scheduling: From Theory to Practice

THÈSE DIRIGÉE PAR :

T'KINDT Vincent	Professeur des Universités, Université François Rabelais de Tours

RAPPORTEURS :

ARTIGUES Christian	Directeur de Recherche, LAAS-CNRS, Toulouse
PASCHOS Vangelis	Professeur des Universités, Université Paris-Dauphine

JURY :

ARTIGUES Christian	Directeur de Recherche, LAAS-CNRS, Toulouse
DELLA CROCE Federico	Professeur, Politechnico di Torino, Italie
LENTE Christophe	Maître de Conférences, HDR, Université François Rabelais de Tours
LIEDLOFF Mathieu	Maître de Conférences, HDR, Université d'Orléans
PASCHOS Vangelis	Professeur des Universités, Université Paris-Dauphine
PINSON Eric	Professeur des Universités, Institut de Mathématiques Appliquées (UCO)
T'KINDT Vincent	Professeur des Universités, Université François Rabelais de Tours

Acknowledgment

I really enjoy the writing of this "chapter", which provides me an opportunity to review my work and my life during the PhD study. The first person that appears in my mind, to whom I feel really thankful, is my advisor Vincent T'Kindt. It has been an excellent experience to work with him. He always has many ideas and it was not always easy for me to follow his thoughts during meetings. His ideas, which do not always work (of course!), always inspire. He has always been available when I need guidance. Moreover, he trusts me and allows me to have much freedom during the PhD, without which I would not enjoyed these three years so much. I am not qualified to evaluate other aspects but he is at least an excellent supervisor. Merci Vincent.

I also want to thank all my co-authors, namely Christophe Lenté, Mathieu Liedloff, Federico Della Croce and Michele Garraffa, without whom all my work would not have been possible. Special thanks go to Michele, with whom I worked everyday during several months, struggling to search for a beautiful structure in a tree. We often spent several hours in a meeting room without being aware. It is a very nice souvenir.

Now let me get out of the professional context. My wife Ting sacrificed her career in Beijing in order to come and stay with me during my PhD. We met each other in 2004 and she is half of my life, if not all. I cannot imagine what could I do without her support. My parents have had strong insistence on the importance of my education. I would not be here without their encouragement and support. They are always proud of me and I am also proud of being their son and proud of the fact that they are proud of me. I also want to thank my grandmother, great Buddhist philosopher of the family, who has been leading me since my young age and has helped me to establish my view of life, of world and of values. She taught me to seek happiness not from outside but from inside. By well maintaining his inner peace, one can always stay happy in whatever environment. Though I am not (yet) wise enough to be well enlightened, I've already benefited a lot from her words. I am so lucky to born in such family environment!

Now let's talk about Polytech! The beginning of my adventure in France started at 2011. I thank the Mundus program which "imported" me from China. I still remember the day of interview in Beijing, with Jean-Louis Bouquard and Marjolaine Martin. Later plus Audrey Perez-Prada, we've had many nice moments together. The teachers of Mundus department helped me to learn French and the teachers of DI helped me to become a good engineer on Computer Science, which enabled the possibility of my PhD study.

The PhD study lasts only three years but is a long way, sometimes really exciting but sometime pretty despairing. I am so glad to have many PhD friends around me, with

whom I had many interesting discussions, many happy breaks and many rounds of baby foot. I am often curious to know where will we be 10 years later. I hope we can always keep contact even though the world changes fast.

I thought of making a name list of people who helped me a lot, but honestly that will be too long and incomplete. Therefore, I stop here, being aware that I've never been alone.

Thank you all.

Résumé

Cette thèse synthétise les travaux de recherches réalisés pendant les études doctorales de l'auteur. L'objectif de ces travaux est de proposer des algorithmes exacts qui ont une meilleure complexité, temporelle ou spatiale, dans le pire des cas pour des problèmes d'ordonnancement qui sont $\mathcal{NP}$-difficiles. En plus, on s'intéresse aussi à évaluer leurs performances en pratique.

Trois contributions principales sont rapportées. La première concerne un algorithme du type *Dynamic Programming* qui résout le problème $F3||C_{max}$ en $\mathcal{O}^*(3^n)$ en temps et en espace. L'algorithme est généralisé facilement à d'autres problèmes du type Flowshop, y compris les problèmes $F2||f$ et $F3||f$, et aux problèmes d'ordonnancement à une seule machine tels que les problèmes $1|r_i|f$, avec $f \in \{f_{max}, f_i\}$.

La seconde contribution porte sur l'élaboration d'une méthode arborescente appelée *Branch & Merge* pour résoudre le problème $1||\sum T_i$ en $\mathcal{O}^*((2+\epsilon)^n)$ en temps avec $\epsilon > 0$ arbitrairement petit et en espace polynomial. Le travail se base sur l'observation que de nombreux sous-problèmes identiques apparaissent répétitivement pendant la résolution du problème global. A partir de ça, une opération appelée *merge* est proposée, qui fusionne les sous-problèmes (les noeuds dans l'arbre de recherche) identiques autant que possible. Cette méthode doit pouvoir être généralisée à d'autres problèmes.

Le but de la troisième contribution est d'améliorer les performances en pratique des algorithmes exacts procédant par parcours d'un arbre de recherche. D'abord nous avons aperçu qu'une meilleure façon d'implémenter l'idée de *Branch & Merge* est d'utiliser une technique appelée *Memorization*. Avec la découverte d'un nouveau *paradoxe* et la mis en place d'une stratégie de nettoyage de mémoire, notre algorithme a résolu les instances qui ont 300 tâches de plus par rapport à l'algorithme de référence pour le problème $1||\sum T_i$. Avec ce succès, nous avons testé *Memorization* sur trois autres problèmes d'ordonnancement notés $1|r_i|\sum C_i$, $1|\tilde{d}_i|\sum w_i C_i$ et $F2||\sum C_i$, précédemment traités par T'kindt et al. (2004). Les résultats finaux des quatre problèmes ont montré la puissance de *Memorization* appliquée aux problèmes d'ordonnancement. Nous nommons ce paradigme *Branch & Memorize* afin de promouvoir la considération systématique de l'intégration de *Memorization* dans les algorithmes de branchement comme *Branch & Bound*, en tant qu'un composant essentiel. La méthode peut aussi être généralisée pour résoudre d'autres problèmes qui ne sont pas forcément des problèmes d'ordonnancement.

Abstract

This thesis summarizes the author's PhD research works on the design of exact algorithms that provide a worst-case (time or space) guarantee for $\mathcal{NP}$-hard scheduling problems. Both theoretical and practical aspects are considered with three main results reported.

The first one is about a *Dynamic Programming* algorithm which solves the $F3||C_{max}$ problem in $\mathcal{O}^*(3^n)$ time and space. The algorithm is easily generalized to other flowshop problems including $F2||f$ and $F3||f$, and single machine scheduling problems like $1|r_i|f$, with $f \in \{f_{max}, f_i\}$.

The second contribution is about a search tree method called *Branch & Merge* which solves the $1||\sum T_i$ problem with the time complexity converging to $\mathcal{O}^*(2^n)$ and in polynomial space. The work is based on the observation that many identical subproblems appear during the solution of the input problem. An operation called *merge* is then derived, which merges all identical nodes to one whenever possible and hence yields a better complexity.

Our third contributionaims to improve the practical efficiency of exact search tree algorithms solving scheduling problems. First we realized that a better way to implement the idea of *Branch & Merge* is to use a technique called *Memorization*. By the finding of a new *algorithmic paradox* and the implementation of a memory cleaning strategy, the method succeeded to solve instances with 300 more jobs with respect to the state-of-the-art algorithm for the $1||\sum T_i$ problem. Then the treatment is extended to another three problems $1|r_i|\sum C_i$, $1|\tilde{d}_i|\sum w_i C_i$ and $F2||\sum C_i$ previously addressed by T'kindt et al. (2004). The results of the four problems all together show the power of the *Memorization* paradigm when applied on sequencing problems. We name it *Branch & Memorize* to promote a systematic consideration of *Memorization* as an essential building block in branching algorithms like *Branch & Bound*. The method can surely also be used to solve other problems, which are not necessarily scheduling problems.

Contents

1 Fundamentals of Scheduling and Exact Exponential Algorithms 21

1.1 Scheduling Theory . . . 21

1.1.1 History . . . 22

1.1.2 Applications . . . 23

1.1.3 Classification and Notation . . . 24

1.1.3.1 Job Data . . . 25

1.1.3.2 Field α - Machine Environment . . . 25

1.1.3.3 Field β - Job Characteristic . . . 26

1.1.3.4 Field γ - Optimality Criteria . . . 27

1.1.3.5 Example of notation . . . 27

1.2 Exact Exponential Algorithms (EEA) . . . 27

1.2.1 Motivations . . . 28

1.2.2 Common Algorithmic Frameworks . . . 29

1.2.2.1 Branching . . . 29

1.2.2.2 Dynamic Programming . . . 33

1.2.2.3 Sort & Search . . . 34

1.2.2.4 Divide & Conquer . . . 35

1.2.2.5 Memorization . . . 36

1.2.2.6 Inclusion & Exclusion . . . 37

1.2.2.7 Exponential Time Hypothesis (ETH) . . . 38

1.2.2.8 Other techniques . . . 38

1.2.3 Notes on Parameterized Algorithms . . . 39

1.3 Existing EEA for Scheduling . . . 39

2 Dynamic Programming for Flowshop Scheduling Problems 43

2.1 Introduction . . . 43

2.2 The $F3||C_{max}$ problem . . . 43

2.3 A Dynamic Programming for the $F3||C_{max}$ problem . . . 44

2.3.1 Complexity Analysis . . . 48
2.3.2 Computational results . . . 48
2.4 Generalizations to other flowshop problems . . . 49
2.4.1 The $F3\|f_{max}$ problem . . . 49
2.4.2 The $F3\|\sum f_i$ problem . . . 53
2.5 A general framework and its applications . . . 53
2.5.1 Pareto Dynamic Programming . . . 54
2.5.2 Complexity analysis . . . 55
2.5.3 Illustration for the $1|r_i|\sum f_i(C_i)$ and $1|r_i|\max(f_i(C_i))$ problems . . . 56
2.6 Auxiliary Result: complexity lower bound of $F3||C_{max}$ based on ETH . . . 57
2.6.0.1 From 3SAT to $F3||C_{max}$ . . . 58
2.6.0.2 3SAT→SubsetSum . . . 59
2.6.0.3 SubsetSum →$F3||C_{max}$ . . . 60
2.7 Chapter Summary . . . 61

3 *Branch & Merge* on the Single Machine Total Tardiness Problem 63
3.1 Introduction . . . 63
3.2 A *Branch & Reduce* approach . . . 64
3.2.1 A first *Branch & Reduce* algorithm . . . 65
3.2.2 A second *Branch & Reduce* algorithm . . . 67
3.3 A *Branch & Merge* Algorithm . . . 68
3.3.1 Merging left-side branches . . . 70
3.3.1.1 A working example for Left Merge . . . 77
3.3.2 Merging right-side branches . . . 82
3.3.2.1 A working example for Right Merge . . . 91
3.3.3 Complete algorithm and analysis . . . 95
3.4 Experimental results . . . 99
3.5 Additional Results . . . 101
3.5.1 The complexity of the algorithm of Szwarc et al. (2001) . . . 101
3.5.2 Algorithm TTBM-L . . . 102
3.6 Chapter summary . . . 103

4 The Memorization Paradigm: *Branch & Memorize* Algorithms for the Efficient Solution of Sequencing Problems 107
4.1 Introduction . . . 107
4.2 A general framework for *Memorization* in search trees . . . 109
4.2.1 Branching schemes . . . 109
4.2.2 Search strategies . . . 110

4.2.3 Memorization schemes . . . 110
4.2.3.1 Solution memorization . . . 111
4.2.3.2 Passive node memorization . . . 112
4.2.3.3 Predictive node memorization . . . 114
4.2.4 Decision guidelines . . . 115
4.2.4.1 Forward branching and depth first search strategy . . . 115
4.2.4.2 Forward branching and best first search strategy . . . 116
4.2.4.3 Forward branching and breadth first search strategy . . . 116
4.2.4.4 Decomposition branching and depth first search strategy . 117
4.2.4.5 Decomposition branching and best first search strategy . . 117
4.2.4.6 Decomposition branching and breadth first search strategy 118
4.3 Implementation guidelines . . . 118
4.4 Application to the $1|r_i|\sum C_i$, $1|\tilde{d}_i|\sum w_iC_i$ and $F2||\sum C_i$ problems . . . 121
4.4.1 Application to the $1|r_i|\sum C_i$ problem . . . 122
4.4.1.1 Application of the memorization framework and improved results . . . 123
4.4.2 Application to the $1|\tilde{d}_i|\sum w_iC_i$ problem . . . 125
4.4.2.1 Application of the memorization framework and improved results . . . 126
4.4.3 Application to the $F2||\sum C_i$ problem . . . 128
4.4.3.1 Application of the memorization framework and improved results . . . 129
4.5 Application to the $1||\sum T_i$ problem . . . 131
4.5.1 Preliminaries . . . 132
4.5.2 Application of the memorization framework and improved results . . 134
4.6 Chapter summary . . . 137

List of Tables

1.1 Synthesis of the best known worst-case complexities 41

2.1 Preliminary results of dynamic programming 49
2.2 Application of PDP on sequencing problems 61

3.1 A sample instance 77
3.2 A sample instance 92
3.3 The time complexity of TTBM for values of k from 3 to 20 99
3.4 Results for instances of size 40 100
3.5 Results for instances of size 300 100

4.1 Results of new algorithms on the $1|r_i|\sum C_i$ problem 125
4.2 Results of the new algoritihms on the $1|\tilde{d}_i|\sum w_i C_i$ problem 128
4.3 Results of new algorithms on the $F2||\sum C_i$ problem 131
4.4 Results for the $1||\sum T_i$ problem 136
4.5 Conclusions on the tested problems 137
4.6 Summary of publications 142

List of Figures

2.1 Illustration of Theorem 1 . . . 45
2.2 Partial solutions and *Pareto Front* of a given jobset in criteria space . . . 45
2.3 Example of *critical paths* . . . 46
2.4 Recursive procedure of the *Dynamic Programming* algorithm. . . . 47
2.5 Example of a critical path through 3 machines . . . 50
2.6 An example for Lemma 3 . . . 51
2.7 An example for Lemma 4 . . . 52
2.8 An example for Lemma 4, case 2 . . . 52
2.9 Example for Lemma 6 (k=5) . . . 57
2.10 Scheduling overview of the $F3||C_{max}$ instance reduced from SUBSETSUM . . . 60

3.1 The situation when the longest job ℓ is put in position h . . . 65
3.2 Branching scheme of TTBR1 . . . 66
3.3 The branching scheme of TTBR1 at the root node . . . 69
3.4 Left-side branches merging at the root node . . . 71
3.5 An example ($P_{1,9,2,8}$) for Proposition 3 and Corollary 1 . . . 73
3.6 Merging for an arbitrary left-side branch . . . 74
3.7 A sample instance solved by TTBR1 with left-merging integrated . . . 81
3.8 An failed attempt on merging right-side branches . . . 82
3.9 An example of right-side branches merging for $k = 3$ . . . 83
3.10 Generic right-side merging at the root node . . . 84
3.11 An example ($P_{1,9,2,8}$) for Proposition 5 . . . 85
3.12 The right branches of P_n were modified by the right-merging from P . . . 86
3.13 A sample instance solved by TTBR1 with right-merging integrated . . . 94
3.14 Reduction obtained by merging . . . 96

4.1 Solution Memorization . . . 111
4.2 Passive node memorization) . . . 113
4.3 Predictive node memorization . . . 114

4.4 Decision tree for choosing the memorization scheme 115
4.5 Number of solutions and useful solutions in memory for an instance of $1||\sum T_i$ with 800 jobs 120

Introduction

This thesis summarizes some research works on the design of exact algorithms that provide a worst-case (time or space) guarantee for $\mathcal{NP}$-hard scheduling problems. By *exact*, we mean algorithms that search for optimal solutions of a given optimization problem. By *worst-case guarantee*, we aim to minimize the time or space complexity of algorithms in worst-case scenarios. Finally, by *scheduling*, we restrict our target problems to scheduling problems, especially sequencing/permutation problems.

Even though it is always difficult to clearly separate research domains due to the existence of intersections, we regard this work as a connection of two fields: *Scheduling Theory* and *Exact Exponential Algorithms* (EEA for short).

"Scheduling is a decision-making process that is used on a regular basis in many manufacturing and services industries. It deals with the allocation of resources to tasks over given time periods and its goal is to optimize one or more objectives."(Pinedo, 2008). Scheduling theory can be traced back to early 20's, before it really started to form a standalone field since the 60's. Now it is a well established area within operational research and it plays an essential role in industries.

On the other side, we regard EEA as a field which groups research efforts on designing exact algorithms with worst-case complexity guarantee for solving $\mathcal{NP}$-hard problems. The research is not limited to a specific type of problems (like scheduling), but guided by the ambition to explore the intrinsic hardness of $\mathcal{NP}$-hard problems. Traced back to 1956, this ambition was expressed in the famous letter of Gödel to Von Neumann as the following question: "...It would be interesting to know, ...how strongly in general the number of steps in finite combinatorial problems can be reduced with respect to simple exhaustive search."(Sipser, 1992). Such research can provide an idea on the relative hardness of problems: all $\mathcal{NP}$-hard problems do not seem to have the same hardness. In fact, since the early stage of computer science, some of these problems appeared to be solvable with a lower exponential complexity than others belonging to the same complexity class. For instance, the MIS (*Maximum Independent Set*) problem, which asks to find from a given undirected graph, a largest subset of non-adjacent vertices, can be solved in $\mathcal{O}(2^n)$ time by enumerating all subsets. A series of improvements have been proposed in the literature, which allow to solve the problems in $\mathcal{O}(1.26^n)$ (Tarjan and Trojanowski, 1977), $\mathcal{O}(1.2346^n)$ (Jian, 1986), $\mathcal{O}(1.2278^n)$ (or $\mathcal{O}(1.2109^n)$ with exponential space) (Robson, 1986), $\mathcal{O}(1.2202^n)$ (Fomin et al., 2006), $\mathcal{O}(1.2132^n)$ (Kneis et al., 2009), $\mathcal{O}(1.2114^n)$ (Bourgeois et al., 2010) and most recently $\mathcal{O}(1.1996^n)$ (Xiao and Nagamochi, 2017) time. In contrast, the famous TSP (*Traveling Salesman Problem*) requires $\mathcal{O}(n!)$ time to be solved in a brute-force way, when the

size n is chosen as the number of cities. But it can be solved in $\mathcal{O}(n2^n)$ time by *Dynamic Programming* (Bellman, 1962), which is a considerable improvement. However, since then, during more than 50 years, no faster algorithm has been proposed. The question is, can other problems be solved faster than enumeration, if yes, to what extent? If no, are we in front of the unbreakable wall of the intrinsic hardness of the problem? For a survey on the most effective techniques in designing EEAs, readers are kindly referred to the paper of Woeginger (2003) and to the book by Fomin and Kratsch (2010).

The initiation of this research is motivated by the following observations. First, EEA has been well developed during recent decades. For a number of $\mathcal{NP}$-hard problems, new algorithms have been proposed consecutively, each time improving little by little the worst-case complexity of the considered problem. As a result, the community now possesses a powerful "toolbox" containing a number of problem solving techniques that can provide much guidance on solving a given problem and analyzing their worst-case complexity. In contrast, *Scheduling* as a major field within Operation Research, has not benefited much from the development of EEA. In fact, many scheduling problems have been proved as $\mathcal{NP}$-hard, and then most considerations are oriented to the practical efficiency instead of the theoretical worst-case guarantee of algorithms. For example, we can find a large number of state-of-the-art algorithms that are *Branch & Bound* algorithms with many problem-dependent properties incorporated, tested on random generated instances to prove their computational efficiency but without their worst-case complexity discussed. This may partially because of the lack of techniques and computation power at that time, which made EEAs inefficient in practice. Today, with the active development on EEA, it becomes natural, even essential to revisit $\mathcal{NP}$-hard scheduling problems on searching a worst-case guarantee for scheduling problems. Some work have been done by Lenté et al. (2013), who introduced the so-called class of multiple constraint problems and showed that all problems fitting into that class could be tackled by means of the *Sort & Search* technique. Further, they showed that several known scheduling problems are part of that class. However, all these problems require assignment decisions only and none of them require the solution of a pure sequencing problem.

Positioning of the Thesis

The thesis is positioned on the connection of *$\mathcal{NP}$-hard Scheduling Problems* and *Exact Exponential Algorithms*, aiming at proposing EEAs with better time/space complexity for solving $\mathcal{NP}$-hard Scheduling Problems. We put special focus on the following points.

- The tackled scheduling problems are sequencing/permutation problems, that is, a solution is represented by an ordering of operations.
- Besides of the search of a better theoretical complexity, we are also interested in the practical efficiency of the proposed algorithms.
- The proposed algorithms are expected to be generalizable.

Outline

As already presented, the results on EEAs for $\mathcal{NP}$-hard scheduling problems are pretty limited. In this thesis we add several results on this subject. Several classic scheduling (sequencing) problems are treated, notably the three machine flowshop problem (denoted by $F3||C_{max}$) and the single machine total tardiness problem (denoted by $1||\sum T_i$). On the treated problems, our algorithms yield the currently best complexity guarantee; on the techniques used, our techniques, which could be generalized to other problems, provide new hints on the design of exact algorithms solving scheduling problems.

Chapter 1 presents the fundamentals of EEA and scheduling theory. The historical evolution of the scheduling theory is reviewed, with the main development in each decade summarized, based on the interesting review of Potts and Strusevich (2009). Some well known techniques for constructing EEAs are presented including some results on scheduling problems. Basic concepts that will be used later are introduced. We also add some notes on Parameterized Algorithms, even though it is not considered during the thesis.

Chapter 2 presents our first contribution: a *Dynamic Programming* algorithm which solves the $F3||C_{max}$ problem to minimize the makespan, with a running time guarantee. The algorithm can be easily generalized to some other flowshop scheduling problems including the $F2||f$ and $F3||f$ problems, with $f \in \{f_{max}, \sum f_i\}$, and also some single machine scheduling problems with release date $(1|r_i|f)$. The framework is named as *Pareto Dynamic Programming*. We also discuss the practical efficiency of the algorithm and the complexity lower bound of the $F3||C_{max}$ problem based on ETH (*Exponential Time Hypothesis*).

Chapter 3 reports our second contribution: a new method which solves the $1||\sum T_i$ problem with a worst-case complexity improving existing ones. By analyzing the search tree of a basic *Branch & Reduce* algorithm, constructed based on some well known properties, we observed that many nodes (i.e. subproblems) are identical. By avoiding repetitive solution of identical nodes, a new method called *Branch & Merge* is conceived. Identical nodes are *merged* to one. By carefully designing the structure of merging, we are able to analyze and prove the complexity of the resulting algorithm. The method runs in polynomial space and can be generalized to other problems that verify certain properties.

Chapter 4 extends the work of Chapter 3 by considering the practical efficiency of algorithms. During the study, another powerful technique called *Memorization* drew our interest. We discuss the link between *Branch & Merge* and *Memorization*. We also found some new computational properties of the $1||\sum T_i$ problem, which finally allows to solve instances with 300 more jobs than the state-of-the-art algorithm. Then the same treatment is performed on another three problems $1|r_i|\sum C_i$, $F2||\sum C_i$ and $1|\tilde{d}_i|\sum w_i C_i$. The computational results are provided, compared and analyzed.

Finally, in the conclusion chapter, we discuss the inner link between the involved techniques and provide our perspectives.

Chapter 1

Fundamentals of Scheduling and Exact Exponential Algorithms

1.1 Scheduling Theory

> "Scheduling is a decision-making process that is used on a regular basis in many manufacturing and services industries. It deals with the allocation of resources to tasks over given time periods and its goal is to optimize one or more objectives." —Pinedo (2008)

Scheduling plays an essential role in industries. It is undoubted that in any industrial process, some kinds of scheduling decisions must be made to well organize the process of different tasks. The decision can be an assignment of resources to tasks, an ordering of tasks to be processed or the exact time intervals of tasks during which these tasks are processed, etc. In this sense, scheduling covers a really large variety of problems. Some of these problems could be uncritical or just enough simple (the size of the problem is so small that the optimal solution is obvious), while others are extremely difficult and need much research effort to break through. As an example, consider a problem denoted as $1|r_i|\sum C_i$: a set of tasks need to be processed sequentially on one machine, each job has a release date and a processing time, the objective is to find an ordering of tasks which minimizes the sum of completion time of each job. The problem description being simple, this problem is strongly $\mathcal{NP}$-hard.

Today, scheduling has become a major field within operational research and it also benefits contributions from other angles such as Mathematics, Computer Science and Economics. When and how did all this start? What are the applications of scheduling? How are different scheduling problems classified? What are the common techniques to deal with them? In this chapter we try to make a brief review on scheduling theory and clarify these points.

1.1.1 History

Arguably, scheduling as a research field can be traced back to the early twenties when Gantt, as a management consultant, discusses the organization of work to maximize profit in his book "Work, wages and profit"(Gantt, 1916). He proposed to give the foreman each day a list of jobs to be done and to well coordinate the jobs so that there is no (or least) interference between jobs. [1] However, it is about forty years later, when a collection of millstone papers, among which the seminal paper by Johnson (1954), are published that scheduling started to be considered as an independent research area within operational research. Since then, it has been attracting more and more research attention.

Potts and Strusevich (2009) reviewed the milestones of scheduling over fifty years. In an elegant way, the paper provides the reader a snapshot for each decade. Starting from mid fifties, the first decade witnessed the research contributions on combinatorial analysis of scheduling problems. Various problem dependant properties are incorporated into algorithms to restrict the search space. The most typical property would be the priority or precedence relation between jobs which generally states something like "at least one optimal schedule exist such that job A precedes job B". The most common technique to achieve that is some deduction based on *Pairwise Interchange*. It is interesting to notice that most well-known algorithms proposed during this period are polynomial-time algorithms, even though concepts on computational complexity were not yet clear at that time. The edited collection of Muth and Thompson (1963) and the research monograph by Conway et al. (1967) helped on the expansion of the scheduling community.

Works on combinatorial analysis continued during the second decade but the resulting properties are more complex, for example a precedence relation may only be valid under certain conditions. Not being directly useful to solve the problem, those properties induce dominance conditions and serve particularly to build *Branch & Bound* algorithms. Problems that are typically treated by *Branch & Bound* during this period include flowshop scheduling and single machine total weighted tardiness problem. A new book by Baker (1974) stepped onto the stage and replaced the one of Conway et al. (1967) as the main scheduling textbook.

Two topics were highlighted during the third decade: computational complexity and classification scheme. Since the seminal paper of Edmonds (1965), polynomial algorithms have been generally considered as "good" algorithms and therefore became the search target of schedulers. However, it seems impossible to design this kind of algorithm for certain problems, due to their "inherent hardness". This feeling was justified by Cook (1971) which stated the existence of "hard" problems for which polynomial algorithms are unlikely to exist. Karp (1972) followed up by formulating this discovery in a more formal way and by adding the first list of $\mathcal{NP}$-complete problems. Further, the monograph of Garey and Johnson (1979) demonstrated various techniques for proving $\mathcal{NP}$-hardness and discussed the complexity issue of 300 problems from 12 areas including scheduling. The book remains a classical text on the topic. About the classification scheme, the most important work was done by Graham et al. (1979), in which the well-known three-field notation $\alpha|\beta|\gamma$ is proposed. Field α describes the machine environment (single machine, parallel machine,

[1] The scenario that Gantt considered is now called *jobshop*

etc); field β specifies problem-dependent parameters (release time, due dates, etc) and field γ is the objective functions to minimize. More details on this can be found in section 1.1.3.

During the fourth decade, the complexity status of many classic scheduling problems are known. Besides of a more frequent appearance of *Column Generation*, it was still lack of weapons to attack $\mathcal{NP}$-hardness in an exact way. More efforts were put on approximation and heuristic algorithms. Apart from specific algorithms depending on problem properties, various approaches were well discussed. Local Search, a method to improve solution quality by searching the *neighborhood* space, which can be traced back to late 50's (Croes, 1958), starts to attract the attention of schedulers (Nicholson, 1967). The early publications on Simulated Annealing (Kirkpatrick et al., 1983; Černỳ, 1985) made Local Search a charming research area, after which a number of new developments start to pop up. Without being exhaustive, Tabu Search is introduced by Glover (Glover, 1986, 1989, 1990). At the same period, Genetic Algorithms (Goldberg, 1989) also started to gain attention. Ant Colony Optimization was introduced in early 90's by Dorigo et al. (1991); Dorigo (1992); Dorigo et al. (1996), and a survey on this metaheuristic is provided by Dorigo and Blum (2005). A recent survey on main metaheuristics is provided by Boussaïd et al. (2013).

The development of scheduling afterwards (the 00's) was veritably diverse. Classic scheduling models were largely extended by adding new constraints coming from realistic industrial environments. Enhanced models include, for instance, *Online Scheduling* (Pruhs et al., 2004) in which jobs to schedule arrive over time; *Scheduling With Batching* (Potts and Van Wassenhove, 1992; Potts and Kovalyov, 2000) in which a set of jobs can be processed as a batch; *Scheduling With Machine Availability Constraints* (Lee, 1996; Sanlaville and Schmidt, 1998; Schmidt, 2000; Lee, 2004), in which the machines may not be available during some intervals of time due to the need of reparation or maintenance. *Multicriteria Scheduling* (T'kindt and Billaut, 2006), *Robust Scheduling* and *Scheduling with Variable Processing Time*, etc, are also attracting research efforts.

Today (August 2017), searching the term "scheduling problem" in Google Scholar for publications published since 2010 gives more than 530000 results, which affirms that *Scheduling* is now a well established domain. More and more new results and solving tools are being added to this field, while the challenges in front of us remain numerous.

1.1.2 Applications

It is not exaggerated to say that scheduling is involved in all industrial activities and it has a critical role in the decision making process. Indeed, in any well established industry, it is much natural to have decisions to make, on the allocation of resources, schedule of activities, etc. According to the application areas, we may encounter scheduling problems from manufacturing production, project management, computer systems, transportation and distribution and those from service industries, without being exhaustive.

Some application examples are introduced by Pinedo (2008), such as *Airport Gate Assignment Problem*, can be considered as in service industry, in which planes must be assigned to boarding gates according to their arrival and departure time to minimize delays suject to a number of known constraints (e.x. some gates are only accessible for small planes) and also uncertainties (weather, technical problems); *CPU Task Scheduling Prob-*

lems, which are encountered during the evolution of computers and is about scheduling task execution on CPU to maximize system performance and user experience. Other applications include Nurse Rostering Problems in hospitals, University Timetabling, Vehicule Routing Problems, Aircraft Scheduling Problems and Crew Scheduling Problems, Bus and Train Driver Scheduling, Sports Scheduling Problems, etc (Leung, 2004). Also, Harjunkoski et al. (2014) have done an excellent review on the industrial aspect of scheduling: they present briefly the approaches that are adopted by industries, discuss their strength and weakness and also the gap between academic research models and real industrial problem environments.

1.1.3 Classification and Notation

Scheduling problems can be classified according to different characteristics. For instances a given scheduling problem can be:

- *deterministic* or *stochastic*: a scheduling problem is deterministic if all the data (processing times, release dates, etc) of the problems are well defined and it is stochastic in contrary if some of these characteristic values are not known precisely but are known to follow some probability laws.

- *static* or *dynamic*: a scheduling problem is static if all the data are known and will not change during the solution, while it is dynamic if the data may change during the solution and the schedule has to be adjusted in real time. The data changes could be, for instances, new jobs arrive during the solution or the actual processing time of jobs are different than what was expected, etc.

- *unitary* or *repetitive*: a scheduling problem is repetitive if its operations appear to be cyclical and it is unitary if each operation corresponds to one unique product.

A more common way to classify scheduling problems is based on the machine environment and job characteristics. This is well adopted especially in the context of machine and processor scheduling problems. In the following, we present this classification scheme together with the notation, knowing that we focus on machine scheduling problems in the scope of this thesis, hence specific classifications or notations are not provided on some other scheduling areas like Resource-Constrained Project Scheduling Problem (RCPSP), even though the notation presented below can be extended to this problem (Brucker et al., 1999).

As already mentioned in the previous section, the most important work on the classification, especially the notation, of scheduling problems was done by Graham et al. (1979) in which a three-field notation $(\alpha|\beta|\gamma)$ was proposed. This notation has become a well-known language of schedulers, called "Scheduleese" by E.L. Lawler according to Lenstra (1998). The adoption of this notation makes the formulation, representation and comparison of scheduling problems very clear and concise.

The detail of this notation can be found directly in the original paper of Graham et al. or in some classic scheduling textbooks like the one of Brucker (2007). However, for

the completeness of the content of thesis, we present this classification scheme through the sections below for reference purpose.

The generic sequencing problem can be viewed as to process n jobs J_i, $i = 1, ..., n$ over m machines M_j, $j = 1, ..., m$. A *schedule* determines for each job, the time intervals and machines for its execution. We normally suppose that each machine can only execute one job at a time and one job can only be processed by one machine at a time. For instance, the $1|\tilde{d}_i|\sum w_i C_i$ problem asks to schedule jobs on one machine in order to minimize the sum of weighted completion time of jobs, subject to the deadlines of jobs. Note that the solution is often expressed as an ordering of jobs, because jobs are often supposed to start as early as possible in most situations, hence the time intervals of jobs are determined automatically. This is the case for all problems treated in this thesis.

1.1.3.1 Job Data

A given job J_i may consist of multiple operations which are noted as O_{ij} (the j-th operation of J_i). $\mu_{ij} \subset \{M_1, ..., M_m\}$ defines the set of candidate machines that can process O_{ij}. The most common data associated to J_i are the following:

- p_{ij}, the processing time of O_{ij}. We simply note p_i if J_i implies only one operation,
- r_i, the release date of J_i, before which J_i cannot be processed,
- d_i, the due date of J_i job. J_i is then expected to be expected before this date, otherwise some penalty is induced depending on the problem,
- w_i, a weight associated with J_i. This is may be used to define the contribution to the cost function of J_i,
- $f_i(t)$, the cost function which measure the cost induced by J_i when it is completed at time t.

Other data may be present according to the problem under consideration.

1.1.3.2 Field α - Machine Environment

The field α may contain two parameters : $\alpha = \alpha_1\alpha_2$, with α_1 defining the machine environment and α_2 indicating the number of machines. α_2 can be an integer value or m which means an arbitrary fixed number or empty which means an arbitrary number. We have $\alpha_1 \in \{1, P, Q, R, G, X, O, J, F\}$ with the following meaning:

- if $\alpha_1 = 1$, then all jobs must be processed on one single machine, hence $\alpha_2 = \emptyset$,
- if $\alpha_1 \in \{P, Q, R\}$, then each job has a single operation
  - if $\alpha_1 = P$, then we have identical parallel machines, i.e. the processing time of J_i on M_j is $p_{ij} = p_i$, $\forall j = 1..m$,
  - if $\alpha_1 = Q$, then we have uniform parallel machines, i.e. $p_{ij} = p_i/s_j$ with s_j the processing speed of M_j, $\forall j = 1..m$,

  - if $\alpha_1 = R$, then we have unrelated parallel machines, i.e. $p_{ij} = p_i/s_{ij}$ with s_{ij} the speed of M_j for processing J_i, $\forall j = 1..m$,

- if $\alpha_1 \in \{G, X, O, J, F\}$, then each job J_i is made up of multiple operations $O_{i1}, ..., O_{in_i}$, each operation being processed by a single dedicated machine. There may also have precedence relations between operations of the same job. This model in general is called General Shop by setting $\alpha_1 = G$

  - if $\alpha_1 = J$, then we have a jobshop problem, in which we have precedence relations of the form of $O_{i1} \rightarrow O_{i2} \rightarrow ... \rightarrow O_{in_i}$. Also, we often assume that consecutive operations are not processed on the same machine.
  - if $\alpha_1 = F$, then we have a flowshop problem, which is a special jobshop in which $n_i = m$ and $\mu_{ij} = M_j$, $\forall j = 1..m$.
  - if $\alpha_1 = O$, then we have an openshop problem, which is a flowshop without precedence relations between operations of the same job,
  - if $\alpha_1 = X$, then we have a mixed shop problem which combines jobshop and openshop.

1.1.3.3 Field β - Job Characteristic

The field β defines job characteristics or extra problem parameters. Most common values are listed below, and note that multiple values can appear together.

- Preemption. If *pmtn* appears in β, preemption is allowed, i.e. a job being processed may be interrupted and resumed later,

- Precedence. If *prec* appears, there are precedence relations between jobs that can be represented as a directed acyclic graph $G = (V, A)$ with V the job set and A the arc set whose direction indicate the precedence relation. Replace *prec* by other keywords to provide more precise indication : $intree(outtree)$ if G is a tree and the maximum outdegree(indegree) of vertices less or equal to one; $sp - graph$ if G is series parallel.

- Release date. If r_i appears, there is a release date for each job J_i, otherwise $r_i = 0$ for all i,

- Restriction on processing times. For example $p_i = p$ indicates that jobs have equal processing times.

- Due date. If d_i appears, a due date is set for each job J_i, such that J_i is expected to finish at that time, otherwise some cost may arise depending on the problem.

- Deadline. If $\tilde{d}_i$ appears, a deadline is set for each job J_i so that J_i must be finished before it.

- Batching. In some problems jobs can be batched for processing. A setup time is needed to prepare a batch, and the processing time of the batch can be the maximum (or sum of) processing time of jobs in the batch, indicated by $p - batch$ ($s - batch$).

1.1.3.4 Field γ - Optimality Criteria

The criteria to optimize usually depends on the completion time of jobs. We denote the completion time of J_i by C_i and associate a cost function $f_i(C_i)$ to each job. Two bottleneck objectives functions can be met:

$$f_{max} = \max_i f_i(C_i)$$

and sum objectives

$$\sum f = \sum_i f_i(C_i).$$

Various job cost functions are considered in the literature, such as:

- Earliness, denoted by $E_i = max(0, d_i - C_i)$,
- Tardiness, denoted by $T_i = max(0, C_i - d_i)$,
- Lateness, denoted by $L_i = C_i - d_i$,
- Unit penalty, denoted by U_i. With $U_i = 0$ if $C_i < d_i$ otherwise $U_i = 1$,
- Lateness, denoted by $L_i = C_i - d_i$,
- Absolute deviation, denoted by $D_i = |C_i - d_i|$,
- Squared deviation, denoted by $S_i = (C_i - d_i)^2$.

Job weights w_i are often considered in f_i. Therefore, some widely considered objective functions are C_{max}, L_{max}, $\sum T_i$, $\sum w_i T_i$, $\sum U_i$, $\sum w_i U_i$, $\sum D_i$, $\sum w_i D_i$, $\sum S_i$, $\sum w_i S_i$, $\sum E_i$, $\sum w_i E_i$. Note that more than one functions can appear in this field in case of *Multicriteria Scheduling Problems*, in which case the notation is accordingly adapted (T'kindt and Billaut, 2006).

1.1.3.5 Example of notation

As an example, the $F3||C_{max}$ problem asks to schedule jobs in a Flowshop environment with three machines to minimize the global completion time. The $1||\sum T_i$ problem asks to schedule jobs on one machine to minimize the sum of tardiness of jobs, with respect to the due date of each job.

1.2 Exact Exponential Algorithms (EEA)

When considered as an independent research area, EEA is at the intersection of several well established domains such as Combinatorial Optimization, Operations Research, Computer Science, and Complexity Theory, in which we are faced with a *hard* decision or optimization problem for which we search for an exact algorithm. In the context of Complexity Theory, we consider a problem as *easy* if it can be solved in polynomial time, the

algorithms we search for here are hence super-polynomial and most of the time exponential in the input size. If we note the complexity of these algorithms as $\mathcal{O}^*(c^n)$ with c a constant, then the general objective in EEA field is to propose algorithms with c as small as possible. Note that we use extensively the complexity notation $\mathcal{O}^*(\cdot)$ to suppress polynomial multiplicative factors for simplification. That is, $f(x) = \mathcal{O}^*(g(x)) \Leftrightarrow f(x) = \mathcal{O}(p(x)g(x))$, with $p(x)$ a polynomial of x. Recall that, $f(x) = \mathcal{O}(g(x))$ if and only if there exists $c > 0,\ n_0 > 0$, such that $\forall x > n_0,\ 0 \leq f(x) \leq cg(x)$. Other conventional complexity notations like $\Omega(\cdot)$ and $o(\cdot)$ are also adopted. $f(x) = \Omega(g(x))$ if and only if $g(x) = \mathcal{O}(f(x))$. $f(x) = o(g(x))$ if and only if $\forall c > 0,\ \exists n_0 > 0$, such that $\forall x > n_0,\ 0 \leq f(x) < cg(x)$. More details can be found in the classic book of Cormen et al. (2009), chapter 3.1.

Why would these exponential algorithms be interesting for us ? What are the algorithm design techniques used in this domain? We discuss these points through the following subsections.

1.2.1 Motivations

The motivation for the search of *exact* algorithms does not need justification: given an optimization problem, we are supposed to find an optimal solution. The question is why *exponential* ? Note that except explicit statement, in the context of this thesis the complexity of an algorithm refers to the worst-case complexity (time complexity by default). This is a common practice (Cormen et al., 2009) since the worst-case complexity provides an upper bound on the running time of the algorithm, and hence ensures that the performance will not be worse. On the other hand, worst-case occurs pretty often in practice for some problems such as searching an entry that does not exist in a database.

Edmonds (1965) argued on the meaning of a *good* algorithm and he claimed that a *good* algorithm should have its performance algebraically related to the input size of the problem, i.e. the complexity of the algorithm should be polynomial on the input size. At the same period, the complexity class $\mathcal{P}$ started to be considered (Cobham, 1965). Since then, it is well accepted by the community that the search of polynomial time algorithm is the primary objective when solving any problem. Unfortunately, under the well believed $\mathcal{P} \neq \mathcal{NP}$ conjecture, polynomial algorithms do not exist for some *hard* problems, which are often classified as $\mathcal{NP}$-complete or $\mathcal{NP}$-hard problems. EEA are proposed for these problems.

Therefore, first of all we hope it clear that we are searching for Fast Exact Algorithms, however the intrinsic difficulty of target problems implies the exponentiality of resulting algorithms. That is, if we want to find the optimal solution of a *hard* optimization problem with a worst-case guarantee, we are necessarily on the road of searching an EEA, with no other choices. This is the first motivation.

Consider polynomial algorithms as *fast* and exponential algorithms as *slow* is fully justified from an asymptotic view, where the input size is necessarily large. However, it is not the case in practice where the input size is often bounded. It is easy to see that $n^3 > 1.0941^n \cdot n$ for $n \leq 100$. It is therefore worthy to consider exponential algorithms and is particularly interesting to derive *moderately* exponential algorithms. As a canonical example, the MIS (Maximum Independent Set) problem which asks to find a largest subset

of pairwise non-adjacent vertices from a given undirected graph, has been benefiting from a series of algorithmic improvements. To the authors' knowledge, the currently fastest algorithm runs in $\mathcal{O}^*(1.1996^n)$ (Xiao and Nagamochi, 2017), which can be thought as *moderately exponential*. Therefore, the performance of an exponential algorithm in practice is not necessarily bad. This is the second motivation.

As stated before, the high complexity of algorithms comes from the intrinsic difficulty of problems. On the contrary, the research on faster EEA helps understand better the origin of that difficulty and the possibility to tackle it. Another important observation is that all $\mathcal{NP}$-hard problems are not equally hard. Some problems have been shown to have algorithms running much faster than the brute-force (e.g. MIS), while for others (e.g. Satisfiability) the enumeration of all possible solutions stays the best choice. Therefore an enhanced research on EEA for these problems will also lead to a finer complexity classification of them. These theoretical interests are the third motivation.

Finally, for a problem that can be solved in $\mathcal{O}(c^n)$ with c a constant, improving the complexity to $\mathcal{O}(c-\epsilon)^n$ allows to enlarge the size of solvable instances within a given amount of time by a multiplicative factor, while running the $\mathcal{O}(c^n)$ algorithm on a faster computer will only increase this size by an addictive factor. Therefore research efforts are worth to be paid to minimize "c".

1.2.2 Common Algorithmic Frameworks

The ambition to conquer $\mathcal{NP}$-hard problems by designing fast exact algorithms have been existing for a long time. However, it is during recent years that EEA has been attracting more and more research intelligence than ever. Fomin and Kratsch in their monograph *Exact Exponential Algorithms* (Fomin and Kratsch, 2010) attributes the source of this prosperity to the survey of Woeginger (2003), in which several techniques are summarized and several open problems are proposed, intriguing a number of researchers to enter this area. The monograph of Fomin and Kratsch (2010) is a standard textbook on this topic. Most common algorithmic frameworks are well discussed in the book, making it a handy manual when tackling hard problems. In this section, we review some of the most important algorithmic frameworks that are to be considered when facing with an $\mathcal{NP}$-hard problem and we kindly refer the reader to the book (Fomin and Kratsch, 2010) for more details.

1.2.2.1 Branching

Branching is perhaps the most natural solving approach when dealing with an optimization problem, because it is based on the simple enumeration idea. All combinatorial problems can be solved in finite time by enumerating the whole solution space. A candidate solution can be represented as a set of variables and the optimal solution has its variables' values inducing the optimal objective function value. A naive branching algorithm, at each *branching*, simply chooses a variable and fix its value - we say that we branch on this variable - to create a new *branch*, which corresponds to a subproblem with one less variable to consider. The algorithm continues in this way and get a candidate solution when all variables are fixed. And the algorithm stops when all possible values of all variables have been tested. The optimal variable assignment is returned, representing the optimal

solution of the problem. Based on this initial structure, the algorithm can be enhanced in different ways to avoid enumerating the whole solution space, by making use of problem dependent or independent properties.

As an example, consider the MIS (Maximum Independant Set) problem. Given an undirected graph $G = (V, E)$, it asks to find $S \subseteq V$ with the maximum cardinality, such that $\forall u, v \in S$, $(u, v) \notin E$. A trivial brute-force algorithm or a naive branching algorithm would just try all $2^{|V|}$ subsets of vertices, verify the constraint of non-adjacency and return the maximum subset found. However, a simple utilization of the non-adjacency property allows to drastically reduce the search space. Consider branching on a vertex v for two possibilities (hence implying a boolean variable): either v belongs to a maximum independent set S or not. Two observations follow:

1. $v \in S$: according to the non-adjacency property, all neighbors of v, denoted by $N[v]$, must be excluded from S.

2. $v \notin S$: then at least two neighbors of v are inside S. Otherwise by forcing v into S and remove $N[v]$ from S, we have a not-smaller independent set.

The above observations can be embeded into the branching procedure: whenever we make decision on one vertex v, we are also making decisions or partial decisions on some other vertices. This boosts the branching algorithm since it is much faster than the pure enumeration on all vertices. Similar but more complex observations on this problem can further enhance the branching.

The above branching algorithm is referred to as *Branch & Reduce* by Fomin and Kratsch (2010). Note that several different names are used in the literature such as search tree algorithm, backtracking algorithm, Davis-Putnam type algorithm, etc. Branching algorithms can be usually represented over a search tree, in which each node represents a subproblem to be solved, which is explored by the algorithm. Various problem dependent or independent techniques can be applied to *prune* the tree, that is to reduce the search space and hence accelerate the solution. Depending on the pruning scheme, we face various names of branching algorithms like *Branch & Bound*, which is commonly used to solve $\mathcal{NP}$-hard problems. Considering a minimization problem, at each node, the algorithm evaluates a LB (Lower Bound) on the objective function value that we can expect if branch on this node. If the LB is greater than the global UB (Upper Bound) given by a candidate solution, then the current node should be cut since it will not provide a better result. Note that even though *Branch & Bound* is widely used in practice, it is not commonly considered in the context of EEA. This is due to the difficulty to analyze tightly its worst-case complexity, which is usually determined by the chosen branching scheme. In fact, the practical average efficiency of *Branch & Bound* algorithms largely depends on the quality of the LB computed at each node. On the theoretical side, proving that this pruning scheme removes an exponential number of nodes, in the worst-case scenario, is a hard task. Other techniques that can be categorized as branching algorithms include *Branch & Cut*, *Branch & Price* etc. Integer Programming and Constraint Programming can also be seen as a problem solving approach using internal branching algorithm. Again, we insist on the fact that in the context of EEA, we are only interested in branching algorithms that can provide a worst-case running time guarantee better than that of brute-force algorithms.

Complexity Analysis

The complexity analysis of branching algorithms is worthy to be introduced here. Since branching algorithms are usually represented as a tree and each single branching operation is usually done in polynomial time, the time complexity of branching algorithms depends on the total number of nodes or the number of leave nodes in the tree. Let $T(n)$ be the maximum number of leaves in a branching tree of input size n. A recurrence relation can be established between a node and all its subnodes if the branching rule is systematic. Let us show this on the above presented MIS problem, without considering the observation 2. Let $deg(v) = |N[v]| - 1$ be the degree of v in G and $deg(G) = \max_{v \in V} deg(v)$. At each branching, we need to decide whether a vertex v belongs to a maximum independent set S or not. If yes, a subnode is created which corresponds to a problem of size $(n - |N[v]|)$, with $N[v]$ denoting the set of neighbours of v (including v), according to observation 1; otherwise if v is excluded from S, a subnode is created corresponding to a subproblem with v deleted from the graph, therefore of size $(n - 1)$. The following relation is then established:

$$T(n) \leq T(n - |N[v]|) + T(n - 1)$$

If we decide to choose v as the vertex with maximum degree, then $deg(v) \geq 3$ because if $deg(v) \leq 2$ we can solve it in polynomial time without need to branch (Fomin and Kratsch, 2010). Hence, the recurrence becomes

$$T(n) \leq T(n - 4) + T(n - 1)$$

This kind of linear recurrences are known to have the solution of the form $T(n) = c^n$, with c the largest root of the equation

$$x^n = x^{n-4} + x^{n-1}$$

.

More generally, we may associate to this recurrence a *branching vector*, which is defined as the vector of the size reduced in each branching case, hence here $\langle 4, 1 \rangle$. Given a branching algorithm, we analyze its induced branching vector in worst-case, then solved the related equation of the above form to obtain the complexity. The equation can be solved easily by some mathematical solvers, (the one used here is WolframAlpha[2]) and the returned result is $T(n) = \mathcal{O}(1.3803^n)$. Notice that this complexity is in fact an upper bound on the worst-case time complexity of the branching algorithm. Recurrence solving is well discussed in textbooks on discrete mathematics (Graham et al., 1994; Rosen, 2012).

Measure & Conquer

The above complexity analysis supposes that every vertex has the same weight (which is 1) in the problem size measurement. This explains the branching vector $\langle 4, 1 \rangle$. Nevertheless, the analysis is not necessarily tight in the sense that the vertex v may have $deg(v)$ much greater than 3, but this cannot be considered in the recurrence function. More concretely,

[2] `https://www.wolframalpha.com/`

branching on a vertex v with $deg(v) = 3$ versus branching on vertex u with $deg(u) = 100$ makes a huge difference on the running of the algorithm, though both vertices are considered having the same weight (which is 1) in the previous analysis. Also, as stated above, a vertex v with $deg(v) \leq 2$ do not need to be branched, hence, it has a little impact on the running time, but they are also holding the same weight (which is 1) as others.

With all these observations in mind, a tighter complexity analysis method is proposed as *Measure & Conquer*. The key is to choose the problem measure in a clever way. Three conditions need to be satisfied:

1. The measure of a subproblem obtained by a branching must be smaller than the measure of the father problem.
2. The measure of an instance is non-negative.
3. The measure of the input should be upper bounded by some functions of natural parameters of the input. The "natural parameter" here means the classic measure choices such as the number of vertices in a graph. This consideration allows to compare the complexity result obtained by Measure & Conquer to classical results.

Consider the following algorithm :

1. If $\exists v \in V$, $deg(v) \leq 1$, count v into the maximum independent set S
2. If $deg(G) = 2$, solve the problem in polynomial time.
3. If $deg(G) \geq 3$, branch on v with $deg(v) = deg(G)$.

By the classical analysis, the branching vector is $\langle 1, 4\rangle$, and the result is $T(n) = \mathcal{O}(1.3803^n)$ as presented above. Now consider a new weight attribution:

1. $\forall v$ with $deg(v) \leq 1$, $weight(v) = 0$, since these vertices do not need to be branched and they are treated in polynomial time.
2. $\forall v$ with $deg(v) = 2$, $weight(v) = 0.5$
3. $\forall v$ with $deg(v) \geq 3$, $weight(v) = 1$

Now we analyze the situation when branching on v. As a starting point, consider $deg(v) = deg(G) = 3$, if v is selected into S, then the total weight of instance decreases by 1 for the removal of v. The neighbors of v, i.e. $N(v)$ should also be removed and their initial degree is at least 0.5, therefore the total weight reduction is $1 + 0.5 * 3 = 2.5$. If v is not selected in S and removed from V, then the degree of u, $u \in N(v)$, passes from 1 to 0.5 or from 0.5 to 0, therefore the total weight reduction is $1 + 0.5 * 3 = 2.5$. The branching vector becomes $\langle 2.5, 2.5\rangle$, inducing $T(n) = \mathcal{O}^*(1.3196^n)$. What about the case where $deg(G) \geq 4$? Selecting or discarding v implies the removal of v which decreases the weight by 1. The weight decrease on the neighbours of v depends on their initial weight. Therefore let $x = |\{u : u \in N(v) \bigwedge d(u) = 2\}|$, $y = |\{u : u \in N(v) \bigwedge d(u) = 3\}|$ and $z = |\{u : u \in N(v) \bigwedge d(u) \geq 4\}|$. Now the following relations hold:

$$\begin{cases} x + y + z = d(v) \\ dis = 1 + 0.5x + 0.5y \\ sel = 1 + 0.5x + y + z \end{cases}$$

According to Lemma 2.3 in the book of Fomin and Kratsch (2010), the branching factor is smaller when it is more balanced, i.e. $\langle k, k \rangle \leq \langle i, j \rangle$ if $i + j = 2k$. Hence $\langle dis, sel \rangle \leq \langle 1, dis + sel - 1 \rangle = \langle 1, 1 + x + 1.5y + z \rangle \leq \langle 1, 1 + d(v) \rangle \leq \langle 1, 5 \rangle$.

That is, the branching vector is at least $\langle 1, 5 \rangle$, inducing $T(n) = \mathcal{O}^*(1.3248^n)$. Consequently, without modifying a single part of the algorithm, we have proved that the complexity is bounded by $\mathcal{O}^*(1.3248^n)$ instead of $\mathcal{O}^*(1.3803^n)$.

For more details on Measure & Conquer, we refer the reader to the works of Fomin et al. (2005a, 2006, 2009).

1.2.2.2 Dynamic Programming

Since its early appearance in the 50's (see for instance Bellman (1952, 1957)) with the term coined by Richard Bellman (Dreyfus, 2002), *Dynamic Programming* (DP) has become rapidly one of the most important algorithmic tool for solving complex optimization problems; especially for problems whose solution can be expressed as a sequence of decisions. The elementary principle behind DP is often referred to as *Principle of optimality* which can be briefly stated as follows:

> *The partial solutions of an optimal solution are themselves optimal for the corresponding subproblems.*

Though it seems not complex to be understood today, this principle directly induces DP as a powerful methodology. For instance, until now the DP is still the algorithm having the best time complexity ($\mathcal{O}^*(2^n)$) for the *Travelling Salesman Problem* (TSP) since the 60's. The corresponding recursive equation, also called Bellman equation, of the DP for the TSP is of the following form (Bellman, 1962; Held and Karp, 1962):

$$Opt(S, i) = \min\{Opt(S - \{i\}, j) + d(j, i) : j \in S - \{i\}\} \tag{1.1}$$

where $S = \{2, \ldots, n\}$, $Opt(S, i)$ denotes the length of the shortest path from city 1 to i passing all cities in S, with $Opt(\{i\}, i)$ initiated to $d(1, i)$, the distance between cities 1 and i. The same idea can be easily extended to general sequencing problems with regular non-decreasing cost functions, as stated by Held and Karp (1962); Woeginger (2003). In fact, when seeking a worst-case guarantee, Dynamic Programming seems the most suitable choice for $\mathcal{NP}$-hard scheduling problems and it yields the best worst-case complexity for a lot of them, even though it has the drawback of having an exponential space complexity. The reader will see later that our first contribution is also about a Dynamic Programming algorithm.

1.2.2.3 Sort & Search

The brute-force approach on $\mathcal{NP}$-hard optimization problems often requires an enormous time because of the complete traverse of the whole solution space. Faster algorithms are able to eliminate part of the solution space during the search without missing the/an optimal solution.

The idea of *Sort & Search* is to rearrange the solution space by some preprocessing on the input data so that the search for an optimal solution is better guided. More concretely, it first sorts partial candidate solutions into some order, then later when constructing a complete global solution by several partial solutions, the construction can be done quickly by means of efficient search.

We illustrate this technique on the Subset Sum problem. In this problem we are given a set of positive integers $a_1, a_2, ..., a_n$ and A, and the aim is to find a a subset $I \subseteq \{a_1, ..., a_n\}$ such that $\sum_{i \in I} i = A$.

The trivial enumeration algorithm traverses all subsets of $\{a_1, ..., a_n\}$ and tests for each one whether its sum gives A. The induced worst-case time complexity is therefore $\mathcal{O}^*(2^n)$. Now consider the following treatment.

1. Partition integers $\{a_1, ..., a_n\}$ into two equal-sized sets S_1 and S_2 in an arbitrary way. Assume n is even without loss of generality.

2. Generate all subsets of S_1 and S_2 respectively. Let us note $SS_1 = \{I | I \subseteq S_1\}$ and $SS_2 = \{I | I \subseteq S_2\}$. Hence, $|SS_1| = |SS_2| = 2^{n/2}$. Note that if $\exists I \subseteq \{a_1, ..., a_n\}$ such that $\sum_{i \in I} i = A$, then $\exists I_1 \subseteq SS_1, I_2 \subseteq SS_2$ such that $I_1 \cup I_2 = I$.

3. Sort elements of SS_2 according to their sum value. (Sort Phase)

4. Enumerate elements of SS_1, for each $I_1 \subseteq SS_1$, search in SS_2 an element I_2 such that $sum(I_2) = A - sum(I_1)$. (Search Phase)

The correctness of the algorithm is evident. What is its complexity? The best complexity of comparative sorting is $\mathcal{O}(N \log N)$ for N elements, therefore the sorting here requires $\mathcal{O}(2^{n/2} \log 2^{n/2}) = \mathcal{O}(n2^{n/2})$. The searching on the sorted set SS_2 for each $I_1 \subseteq SS_1$ requires $\log 2^{n/2}$. Therefore the global time complexity is $\mathcal{O}(n2^{n/2}) + 2^{n/2} \log 2^{n/2} = \mathcal{O}(n2^{n/2})$ which is faster than $\mathcal{O}(2^n)$.

Note that the space complexity is $\mathcal{O}(2^{n/2})$, induced by the storage of SS_1 and SS_2.

Similarly, *Sort & Search* can be applied on Knapsack and Exact Satifiability (Fomin and Kratsch, 2010; Horowitz and Sahni, 1974; Schroeppel and Shamir, 1981). Lenté et al. (2013) proposed an extension of this method to a general class of problems called *Multiple Constraint Problems* and use the method to derive faster exponential-time algorithms for several machine scheduling problems.

Sort & Search accelerates the solution by making use of preprocessed information (sorting) at the price of more space requirement. The relation between time and space complexity of algorithms is further discussed in section 1.2.2.4 and section 1.2.2.5.

1.2.2.4 Divide & Conquer

Divide & Conquer is one of the most common algorithm design paradigm. It can be applied on problems whose global optimal solution can be constructed from several optimal subsolutions of subproblems. At this point, it reminds us about *Dynamic Programming* and as what will be shown later, these two techniques do have strong inner connections.

In *Divide & Conquer*, a problem is solved recursively with three steps applied on each level of recursion (Cormen et al., 2009):

1. **Divide** the problem into a number of subproblems which are instances of the same problem but with smaller size.

2. **Conquer** each subproblem in recursive way and when the size of subproblem is small enough, corresponding to the elementary case, solve it in a straightforward way without recursion.

3. **Combine** the obtained solution of each subproblems to form the optimal solution of the initial problem.

A very natural *Divide & Conquer* example as given by Cormen et al. (2009) is the *Merge Sort* algorithm. Given an unsorted array of numbers, merge sort first divide the array in the middle, sort each half array in a recursive way then combine the two sorted sequence into a global sorted one. The elementary case arises when the array to sort contains one element only.

The *Divide* operation for sorting is pretty straightforward, nonetheless it is not always the case depending on the problem. Here we present an interesting application of *Divide & Conquer* on Hamiltonian Path problem, initially proposed by Gurevich and Shelah (1987). Given an undirected graph $G = (V, E)$, and two vertices $s, t \in V$, $|V| = n$, the problem asks whether there exists a path P from s to t of length $(n-1)$, which pass by each vertex exactly once.

Here is the *Divide & Conquer* thinking: if a Hamiltonian path $P = (s, v_1, v_2, ..., v_{n-2}, t)$ exists, it can be seen as the concatenation of two half-paths $P_1 = (s, v_1, ..., v_{(n-2)/2})$ and $P_2 = (v_{(n-2)/2}, ..., v_{n-2}, t)$. Furthermore, P_1 (P_2) is a Hamiltonian path of a subproblem where the graph is induced by the vertices V_1 (V_2) from the path. This naturally suggests a *Divide & Conquer* approach. However, unlike the sorting problem, here we cannot divide V arbitrarily into V_1 and V_2, since not all divisions can lead to an correct answer. Therefore, we need to try all possible divisions.

Let $HP(V \setminus \{s,t\}, s, t) = true$ if there exists a hamitonian path from s to t passing by all $v \in V \setminus \{s,t\}$. We first choose the middle vertex $v_{(n-2)/2}$ then try all possible partitions of V_1 and V_2, finally combine P_1 and P_2. The following recurrence relation holds:

$$HP(V \setminus \{s,t\}, s, t) = \bigvee_{\substack{V_1 \subset \{V \setminus \{t\} \bigcup \{s\}\}, V_2 \subset \{V \setminus \{s\} \bigcup \{t\}\}, \\ |V_1| = |V_2| = (|V|-3)/2+1, \\ V_1 \bigcap V_2 = \{v_{mid}\}}} \Big(HP(V_1, s, v_{mid}) \bigwedge HP(V_2, v_{mid}, t) \Big)$$

The time complexity induced by the above recurrence is

$$\begin{aligned} T(n) &\leq (n-2)2^{n-3}2T((n-3)/2) \\ &\leq n2^n 2T(n/2) \\ &= 2^{n(1+1/2+1/4+...)}n^{\log n}T(1) \\ &= \mathcal{O}^*(4^n n^{\log n}) \end{aligned}$$

Since the algorithm is recursive, it required only polynomial space. We can apply *Divide & Conquer* on other problems as shown by Björklund and Husfeldt (2008); Bodlaender et al. (2006).

Time, space and Dynamic Programming
The Hamiltonian Path problem can be solved by *Dynamic Programming* in $\mathcal{O}^*(2^n)$ time and space. Nevertheless, it can be unified with *Divide & Conquer* on this problem and other sequencing problems with a similar structure. Indeed, reconsider the above *Divide & Conquer* algorithm, instead of selecting the intermediate vertex for the middle position and trying all 2-partitions of the remaining vertices, we can choose the position of the intermediate vertex arbitrarily. When the intermediate vertex is on the last position, we got a Dynamic Programming algorithm.

This suggests us to consider the compromise between time and space requirement of our algorithm. Actually it is proved that for every $i \in \{0, ..., \log_2 n\}$, the problem TSP on n cities is solvable in time $\mathcal{O}^*(2^{n(2-1/2^i)}n^i)$ and space $\mathcal{O}^*(2^{n/2^i})$ (Fomin and Kratsch, 2010). The idea is to first apply *Divide & Conquer* until reaching subproblems of limited size, which is chosen depending on i, then call Dynamic Programming to solve it.

We suggest to always consider these two techniques as a bundle whenever solving a given problem, so as to achieve a better time-space compromise.

1.2.2.5 Memorization

We have already seen the possibility to achieve a time-space compromise by combining *Dynamic Programming* and *Divide & Conquer* (section 1.2.2.4). Another technique allowing to achieve this is *Memorization*. As its names suggests, memory space is used to store useful information which could accelerate the solution. More concretely, it is applied upon a base algorithm, most usually a branching algorithm, during the execution of which, identical subproblems may be encountered many times. With *Memorization* enabled, each time when a subproblem is solved, the corresponding subsolution is "memorized", that is, stored into a hashtable, a database or any arbitrary data structure allowing a fast query. When the same subproblem is encountered again, instead of solving it again, it can be just retrieved from the "memory".

As an example, consider the branching algorithm for the MIS problem introduced in section 1.2.2.1. For a given graph of n vertices the algorithm generates $\mathcal{O}(1.3803^n)$ subproblems. For any size $1 \leq p \leq n$, a subproblem of size p can be solved in $\mathcal{O}(1.3803^p)$ time, hence the number of subproblems of size p solved during the solution of an instance of size n, denoted by $N_p(n)$, is at most $\frac{T(n)}{T(p)} = \mathcal{O}(1.3803^{n-p})$. With *Memorization*, each

subproblem is solved only once, therefore $N_p(n) \leq \binom{n}{p}$. This leads to

$$N_p(n) = \mathcal{O}(\min(1.3803^{n-p}, \binom{n}{p}))$$

with $N_p(n)$ reaches its maximum when the two inner terms are balanced. The equation

$$1.3803^{n-p} = \binom{n}{p}$$

yields $p = 0.0865n$. Therefore the number of subproblems of size $0.0865n$ is bounded by $1.3803^{n-p} = 1.3424^n$ and the total number of subproblems of all sizes is bounded by $n1.3424^n = \mathcal{O}^*(1.3424^n)$ which is the running time of the algorithm with memorization. The space complexity is also $\mathcal{O}^*(1.3424^n)$ since the solution of all subproblems are memorized when these problems are solved at the first time. More details on this analysis can be found in the work of Fomin and Kratsch (2010); Robson (1986).

Note that even though some algorithms also use memory space to store information for faster solving, such as *Sort & Search*, *Memorization* refers particularly to the use of memory for directly storing solutions of subproblems.

1.2.2.6 Inclusion & Exclusion

The above mentioned techniques share a common property: they do better than the brute-force search by avoiding exploring the whole solution space, making use of some problem dependent/independent properties. There exists some algebraic techniques that tackle problems from other angles. One representative technique is called *Inclusion & Exclusion*. For a given $\mathcal{NP}$-complete problem, instead of searching directly for a certificate, it counts the number of certificates and if this number is greater than zero, the corresponding decision problem gets a YES answer. Counterintuitively, counting the number of certificates is sometimes easier than verifying their existence.

Karp seems the first person illustrating the use of *Inclusion & Exclusion* on solving hard problems. He stated in his paper (Karp, 1982) that many problems of sequencing, packing and assignment can be approached by *Inclusion & Exclusion*. As an example, he showed how to apply it to solve Hamiltonian Path, a generic scheduling problem with release date and deadline and Bin Packing. We summarize here the solution of Hamiltonian Path by *Inclusion & Exclusion*.

Let $G = (V, E)$ be a digraph with $V = \{s, t\} \bigcup \{1, ..., n\}$. Hamiltonian Path asks to determine whether there is a Hamitonian path of length $(n + 1)$ from s to t, passing each vertex $\{1, ..., n\}$ exactly once. Let X be the number of the hamitonian paths. For $S \subseteq \{1, ..., n\}$, let $N(S)$ be the number of paths of length $(n+1)$ from s to t that containing no vertex from S. Then, according to the principle of *Inclusion & Exclusion*, the following relation holds.

$$X = \sum_{S \subseteq \{1,...,n\}} (-1)^{|S|} N(S)$$

For fixed S, $N(S)$ can be computed by adjacency matrix multiplication Fomin and Kratsch (2010). Let A be the adjacency matrix of the induced graph containing vertices in $V \setminus S$, the entry at row i and column j of the matrix A^{n+1} corresponds to the number of paths of length $(n+1)$ from i to j. Note that a vertex can appear repetitively in the path. Since matrix multiplication can be performed in polynomial time, $N(S)$ can therefore be computed in polynomial time. There are 2^n different values of S, thus the time complexity of the *Inclusion & Exclusion* algorithm is $\mathcal{O}^*(2^n)$. Moreover, the space requirement is polynomial. Note that $N(S)$ can also be computed via *Dynamic Programming* (Karp, 1982).

1.2.2.7 Exponential Time Hypothesis (ETH)

Unlike the above presented techniques used to design fast exact exponential algorithms, *Exponential Time Hypothesis* (ETH) is a powerful tool attacking problems from another direction: it is used to prove complexity lower bounds. The conjecture $\mathcal{P} \neq \mathcal{NP}$ implies that $\mathcal{NP}$-hard problems cannot be solved in polynomial time. However, no more precision is given on the complexity of $\mathcal{NP}$-hard problems except that they are super-polynomial. People have realized that this simple binary classification of decision problems to *easy* or *hard* is perhaps too coarse-grained. This encourages to analyze the complexity of problems in a finer way. On this direction, *Exponential Time Hypothesis*, proposed by Impagliazzo and Paturi (2001), states that 3SAT cannot be solved in sub-exponential time ($\mathcal{O}(2^{o(n)})$). Based on this assumption, complexity lower bounds have been deduced for many problems such as k-Colorability, k-Set Cover, Independent Set, Clique, and Vertex Cover. The existence of sub-exponential algorithms for any of them implies the same for the others. Moreover, a *Strong Exponential Time Hypothesis* (SETH) has also been proposed, which assumes that k-CNF-SAT requires $\mathcal{O}(2^n)$ time in the worst-case when k grows to infinity. This allows to derive even tighter bounds such as shown by Cygan et al. (2016): for every $\epsilon < 1$ an $\mathcal{O}(2^{\epsilon n})$ algorithm for Hitting Set, Set Splitting or NAE-SAT would violate SETH.

Therefore, when it seems difficult to derive EEAs with a better complexity, it may because that the complexity lower bound of the problem is reached. It is hence interesting to prove that by using ETH.

1.2.2.8 Other techniques

Besides *Inclusion & Exclusion*, other interesting algebraic techniques exist, among which there is *Fast Subset Convolution*. We kindly refer the reader to the work of Fomin and Kratsch (2010); Björklund et al. (2007) for further details. The use of algebraic methods to decrease space complexity is discussed by Lokshtanov and Nederlof (2010).

Some other techniques that are discussed in the book of Fomin and Kratsch (2010) include algorithms based on *Treewidth*, *Local Search*, etc. We kindly refer the reader to this book for further details.

1.2.3 Notes on Parameterized Algorithms

Parameterized Algorithms are algorithms whose complexity depends not only on the input size of instances but also on some extra parameters. This is a fairly new field of complexity theory, which is strongly related to *Exact Exponential Algorithms*. The latter can be seen as parameterized algorithms which uses the input size as parameter. Therefore, it is necessary to add some notes on Parameterized Algorithms, even though no specific research efforts are paid on this during the PhD study.

Parameterized complexity theory is created as a framework which allows to measure the complexity of algorithms more finely. Conventional complexity theory measures the complexity only in terms of the input size, hence ignores the structural information about the input instances. Consequently, the resulting complexity may make the problem appear harder than it really is. With Parameterized Algorithms, the complexity depends both on the input size and another well chosen parameter, which is expected to be "small".

Analogously to the classic P-NP complexity framework, a complexity hierarchy has been established to classify problems according to their hardness. The "easiest" problems in this hierarchy are *fixed-parameter tractable* (FPT) problems, which are problems that can be solved by algorithms having a complexity of $f(k)p(n)$, where n is the input size, k is the chosen parameter and $p(\cdot)$ is a polynomial function. Since the complexity depends only polynomially on the input size, the running time is hence expected to be small when k is well chosen as a parameter with small value in most instances.

In fact, algorithms have always been analyzed in terms of different parameters, so parameterized algorithms do exist before this well defined framework is established. Nevertheless, the framework is also very important in the other direction, which is to prove the *intractability* of problems. In the above mentioned parameterized complexity hierarchy, FPT problems are easiest ones, analogously to class P in the P-NP system, and class W[P] is an analogue of NP. But unlike the P-NP system, many natural problems are neither in FPT nor W[P]-complete. To classify these problems, a hierarchy of classes within W[P] is established, called W-hierarchy. It is usually presented as

$$FPT \subseteq W[1] \subseteq W[2] ... \subseteq W[P].$$

Parameterized complexity theory was developed by Downey and Fellows in early 90's. Their monograph (Downey and Fellows, 1999) provided a picture on the foundation of this theory and it has been updated in 2013 (Downey and Fellows, 2013). Another referential book in the area is the one of Flum and Grohe (2006). More literature references can be found through these books.

1.3 Existing EEA for Scheduling

Concerning $\mathcal{NP}$-hard scheduling problems, some of the techniques presented in section 1.2.2 have been used to construct exact algorithms with worst-case guarantee.

The most widely applicable technique on scheduling problems is *Dynamic Programming*, which is applied very naturally on sequencing problems (Held and Karp, 1962) since its early occurrence. Its classical application is on TSP which can also be considered as

a scheduling/sequencing problem. Held and Karp (1962) discussed the application of *Dynamic Programming* on general sequencing problems with arbitrary cost function. In the remarkable survey of Woeginger (2003), a single machine scheduling problem under precedence constraints minimizing total completion time is discussed. For all problems discussed in the above two references, the induced time complexity is $\mathcal{O}^*(2^n)$ which is a common result whenever the *Dynamic Programming* is applied in the conventional way, that is, across all subsets. For the latter problem, an algorithm running in $\mathcal{O}((2-\epsilon)^n)$ is reported by Cygan et al. (2014), based on the fact that the *Dynamic Programming* does not need to traverse all subsets: an amount of subsets are infeasible due to the precedence constraints. When applied on more complex problems, the structure and the complexity of the resulting *Dynamic Programming* depends largely on the problem's properties. The job shop scheduling problem can be solved by *Dynamic Programming* in $\mathcal{O}(p_{max}^{2n}(m+1)^n)$ (Gromicho et al., 2012), which is exponentially faster than the brute-force which runs in $\mathcal{O}((n!)^m)$. Jansen et al. (2013b) investigates the lower bounds of complexities of some scheduling and packing problems, based on ETH (*Exponential Time Hypothesis*, see Impagliazzo and Paturi (2001); Fomin and Kratsch (2010)). They showed that problems including Subset Sum, Knapsack, Bin Packing, $P2||C_{max}$ and $P2||w_jC_j$ have a lower bound of $2^{o(n)} \cdot \|I\|^{\mathcal{O}(1)}$, with $\|I\|$ the input length. They also developed a *Dynamic Programming* framework that is able to solve a number of scheduling and packing problems in $2^{O(n)} \cdot \|I\|$ time. Flowshop problems can also be treated by a generalized *Dynamic Programming*, as will be discussed in Chapter 2. The resulting time complexity is $\mathcal{O}^*(c^n)$ with $c > 2$ a constant. Noteworthy, for all problems that can be treated by *Dynamic Programming* in the conventional way, *Divide & Conquer* can usually also be applied which in consequence increases the time complexity but requires less space.

Dynamic Programming being broadly applicable on scheduling problems, it is often considered as not practical due to its exponential space complexity. As reported in Section 1.1.1, the most commonly considered algorithms for solving scheduling problems in practice are *Branching* algorithms, especially *Branch & Bound*. By well choosing the branching scheme and bounding procedures, *Branch & Bound* can often be efficient on solving randomly generated instances. However, when considering the worst case complexity, it is extremely difficult to analyze a *Branch & Bound* algorithm because the effectiveness of the lower bounding procedure depends largely on the actual instance. In other words, it seems very hard to prove the non-existence of a worst-case instance on which all lower bounding procedures are unuseful hence the *Branch & Bound* reduces to a simple enumerative branching algorithm.

The application of *Sort & Search* on scheduling problems is well explored by Lenté et al. (2013). They propose an extension of this method to a general class of problems called *Multiple Constraint Problems* and use the method to derive faster exponential-time algorithms for several machine scheduling problems including $P||Cmax$, $P|d_i|w_iU_i$ and $F2||C_{max}^k$ with a worst-case complexity $\mathcal{O}^*(m^{\frac{n}{2}})$ where m is the number of machines. It is important to notice that the scheduling problems that are tackled by *Sort & Search* always imply solving an assignment problem. However, whether *Sort & Search* can be applied on pure sequencing problems remains unclear.

Inclusion & Exclusion can also be applied to solve some scheduling problems, as initially reported by Karp (1982).

The best known worst-case complexity results are synthesized in Table 1.1.

Problem	Enumeration	EEA	Reference
$1\|dec\|f_{max}$	$\mathcal{O}^*(n!)$	$\mathcal{O}^*(2^n)$	Fomin and Kratsch (2010)
$1\|dec\|\sum f_i$	$\mathcal{O}^*(n!)$	$\mathcal{O}^*(2^n)$	Fomin and Kratsch (2010)
$1\|prec\|\sum C_i$	$\mathcal{O}^*(n!)$	$\mathcal{O}^*((2-10^{-10})^n)$	Cygan et al. (2014)
$1\|prec\|\sum w_iC_i$	$\mathcal{O}^*(n!)$	$\mathcal{O}^*(2^n)$	Woeginger (2003)
$1\|d_i\|\sum w_iU_i$	$\mathcal{O}^*(n!)$	$\mathcal{O}^*(\sqrt{2}^n)$	Lenté et al. (2013, 2014)
$1\|d_i\|\sum T_i$	$\mathcal{O}^*(n!)$	$\mathcal{O}^*(2^n)$	Woeginger (2003)
$1\|r_i,prec\|\sum w_iC_i$	$\mathcal{O}^*(n!)$	$\mathcal{O}^*(3^n)$	Woeginger (2003)
$IntSched$	$\mathcal{O}^*(2^{n\log m})$	$\mathcal{O}^*(1.2132^{nm})$ $\mathcal{O}^*(2^n)$ $\mathcal{O}^*(2^{(m+1)\log_2 n})$	Lenté et al. (2014)
$P\|dec\|f_{max}$	$\mathcal{O}^*(m^nn!)$	$\mathcal{O}^*(3^n)$	Lenté et al. (2014)
$P\|dec\|\sum f_i$	$\mathcal{O}^*(m^nn!)$	$\mathcal{O}^*(3^n)$	Lenté et al. (2014)
$P\|\|C_{max}$	$\mathcal{O}^*(m^n)$	$\mathcal{O}^*(\sqrt{m}^n(\frac{n}{2})^{m+1})$	Lenté et al. (2013)
$P2\|\|C_{max}$	$\mathcal{O}^*(2^n)$	$\mathcal{O}^*(\sqrt{2}^n)$	Lenté et al. (2013, 2014)
$P3\|\|C_{max}$	$\mathcal{O}^*(3^n)$	$\sqrt{3}^n$	Lenté et al. (2013)
$P4\|\|C_{max}$	$\mathcal{O}^*(4^n)$	$\mathcal{O}^*((1+\sqrt{2})^n)$	Lenté et al. (2014)
$P\|d_i\|\sum w_iU_i$	$\mathcal{O}^*((m+1)^n)$	$\mathcal{O}^*(\sqrt{m+1}^n(\frac{n}{2})^{m+2})$	Lenté et al. (2013)
$P2\|d_i\|\sum w_iU_i$	$\mathcal{O}^*(3^n)$	$\sqrt{3}^n$	Lenté et al. (2013)
$F2\|\|C^k_{max}$	$\mathcal{O}^*(2^n)$	$\mathcal{O}^*(\sqrt{2}^n)$	Lenté et al. (2013, 2014)
$J\|\|C_{max}$	$\mathcal{O}^*(n!)$	$\mathcal{O}^*(P^{2n}_{max}(m+1)^n)$	Gromicho et al. (2012)

Table 1.1: Synthesis of the best known worst-case complexities

Recent years, more and more *Parameterized Complexity* results on scheduling problems are appearing, even though among them the majority of results are negative. Without being exhaustive, Bodlaender and Fellows (1995) showed that k-processor scheduling with precedence constraints is W[2]-hard. Fellows and McCartin (2003) state that scheduling unit-length jobs with deadlines and precedence constraints is W[1]-hard when parameterized by the number of tardy jobs. The results of Jansen et al. (2013a) imply that makespan minimization on k parallel machines with jobs of polynomially bounded processing times is W[1]-hard. On the positive side, Mnich and Wiese (2015) propose FPT algorithms for some classic scheduling problems including makespan minimization, scheduling with job-dependent cost functions and with job rejection, by identifying several crucial parameters such as the number of distinct processing times, the number of distinct weights and the maximum number of rejected jobs. They also remark that some scheduling problems can be addressed by choosing as parameter the "number of numbers", introduced by Fellows et al. (2012). Very recently, Van Bevern (2017) reported new parameterized complexity results of scheduling problems with release time and deadlines (Van Bevern et al., 2017), precedence constraints (Van Bevern et al., 2016) and sequence-dependent batch setup times or routing (Van Bevern and Pyatkin, 2016). An up-to-date list of publications on the intersection of *Parameterized Complexity* and *Scheduling* can be found in the community wiki: `http://fpt.wikidot.com/operations-research`.

Chapter 2

Dynamic Programming for Flowshop Scheduling Problems

2.1 Introduction

In this chapter, we present our work on a *Dynamic Programming* algorithm that solves the $F3||C_{max}$ problem. *Dynamic Programming* is a useful tool to tackle permutation problems and it can be applied pretty naturally to solve *decomposable* permutation problems (T'kindt et al., 2004), in which the optimal solution of the problem can be composed by optimal solutions of subproblems. The corresponding *Dynamic Programming* scheme is sometimes referred to as *Dynamic Programming across the subsets* (see Woeginger (2003)). Unfortunately, the $F3||C_{max}$ problem is not *decomposable* since an optimal solution may need to be based on a non-optimal partial solution of a subproblem. This is why we are interested on the study of the application of *Dynamic Programming* to these problems. Our algorithm is easily generalized to some other flowshop problems, among which we provide dedicated complexity analysis on $F3\|f_{max}$ and $F3\|\sum f_i$ problems. After that, we also formulate a generalized framework as *Pareto Dynamic Programming*, which can be applied to other problems (e.g. $1|r_i|f$, $F2||f$, etc) verifying some specific dominance conditions over the *Pareto Front*. This is notably the case of $1|r_i|f$ and $F2||f$ problems.

2.2 The $F3||C_{max}$ problem

The problem tackled in this section and denoted by $F3||C_{max}$, is described as follows. Consider n jobs to be scheduled, each of which must be processed first on machine one, then on machine two and finally on machine three in this order. Each machine can only have one job being processed at a time. We note O_{ij}, $i \in \{1, ..., n\}$, $j \in \{1, 2, 3\}$, the operation of job i on machine j, with p_{ij} a non-negative integer representing the corresponding processing time. The objective is to find an optimal job sequence that minimizes the makespan, which is defined as the completion time of the last job on the last machine in a schedule. Even though there exists a polynomial time algorithm when the number of machines is two, the problem with $m \geq 3$ machine has been proved to be $\mathcal{NP}$-hard (Garey

et al., 1976). When $m = 2$ or $m = 3$, there always exists a permutation schedule which is optimal (Brucker, 2007). A solution is a permutation schedule if jobs are executed in the same order on all machines. Extensive researches have been done on the design of heuristic and metaheuristic algorithms (see Ruiz and Maroto (2005); Framinan et al. (2004)), approximation algorithms (see Smutnicki (1998); Sviridenko (2004)) and also exact methods (see Ladhari and Haouari (2005) and earlier results by Ignall and Schrage (1965); Lomnicki (1965); Brown and Lomnicki (1966); McMahon and Burton (1967); Ashour (1970); Lageweg et al. (1978); Potts (1980); Carlier and Rebaï (1996); Cheng et al. (1997)). For a global view on different approaches on this problem and its numerious variants, we refer the reader to the review of Reza Hejazi and Saghafian (2005).

We focus on the search for an optimal permutation schedule of the problem, with a provable worst-case running time. We provide a theoretical study which aims at establishing upper bounds on the worst-case complexity. Clearly, the problem can be solved trivially by enumerating all possible job permutations, which yields a $\mathcal{O}^*(n!)$ worst-case time algorithm. The current fastest exact algorithms in practice is the *Branch & Bound* algorithm provided by Ladhari and Haouari (2005), which solves randomly generated instances with up to 2000 jobs. However, when considering worst-case scenarios, this branching algorithm can hardly be proved to be faster than the brute-force. An algorithmic framework which solves a number of scheduling problems in time $2^{\mathcal{O}(n)} \times \|I\|^{\mathcal{O}(1)}$, with $\|I\|$ being the input length, is proposed by Jansen et al. (2013b). A dynamic programming algorithm by Gromicho et al. (2012) also exists, solving the job-shop scheduling problem in $\mathcal{O}((p_{max})^{2n}(m+1)^n)$.

The aim of this work is to design a moderately exponential algorithm, that is, an algorithm running in $\mathcal{O}^*(c^n)$ time with the constant c as small as possible, improving the currently best complexity result of $2^{\mathcal{O}(n)} \times \|I\|^{\mathcal{O}(1)}$ (Jansen et al., 2013b). In the next section we introduce a dynamic programming algorithm to solve the $F3||C_{max}$ problem, together with its worst-case complexity analysis.

2.3 A Dynamic Programming for the $F3||C_{max}$ problem

Given a job permutation π, let $C_j^M(\pi)$ be the completion time of π on machine j, $j \in \{1, 2, 3\}$. We omit π whenever it does not create ambiguity. We also define a binary operator "." which concatenates two permutations of jobs.

The proposed *Dynamic Programming* algorithm is based on Theorem 1.

Theorem 1. *Let S be a set of n jobs to be scheduled, and $S' \subset S$ a subset of S. Given π and π' as two permutations of jobs from S', σ as a permutation of jobs from $S \setminus S'$, then we have:*

$$If \begin{cases} C_2^M(\pi) \leq C_2^M(\pi') \\ C_3^M(\pi) \leq C_3^M(\pi') \end{cases} then \begin{cases} C_2^M(\pi.\sigma) \leq C_2^M(\pi'.\sigma) \\ C_3^M(\pi.\sigma) \leq C_3^M(\pi'.\sigma) \end{cases}$$

that is, the partial permutation π dominates π'. Every schedule starting by π' is dominated by a schedule starting by π.

Proof. The proof of Theorem 1 is straightforward and is illustrated in Figure 2.1. First,

note that $C_1^M(\pi) = C_1^M(\pi')$ always holds since π and π' contain the same jobs and there is no idle time between jobs on machine 1. On the second and third machines, π completes earlier (or equal) than π' so σ is always preferred to be put after π instead of π' in order to minimize the makespan C_{max}^M. As shown in Figure 2.1, the schedule containing π is at least as good as the schedule containing π'. □

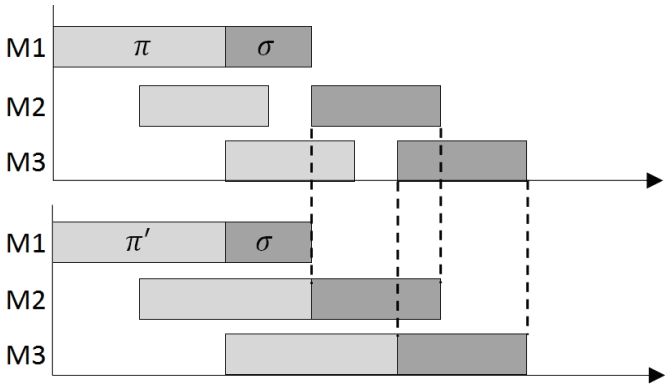


Figure 2.1: Illustration of Theorem 1

The idea of *Dynamic Programming* is the following. When trying to construct an optimal solution from partial solutions, only the partial solutions that are not dominated need to be considered. This can be seen by switching to the criteria space $\langle C_2^M, C_3^M \rangle$ for representation. For a fixed jobset, we plot all its partial permutations as points in the criteria space $\langle C_2^M, C_3^M \rangle$. Then the non-dominated permutations form the *Pareto Front* of points (Figure 2.2).

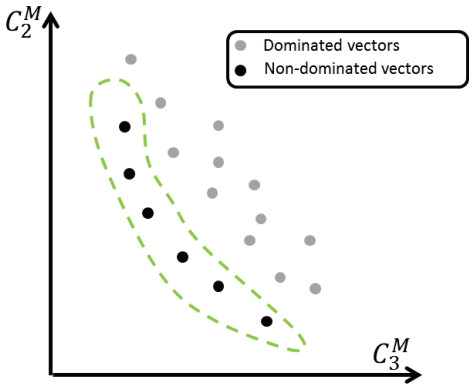


Figure 2.2: Partial solutions and *Pareto Front* of a given jobset in criteria space

Now we formulate the algorithm, starting with some definitions.

Definition 1. *Given a set of permutations of a jobset S, Pareto Permutations define the subset of permutations whose associated criteria vector* $\langle C_2^M, C_3^M \rangle$ *is not dominated by the*

criteria vector of any other permutation from this set. In case that several dominating permutations have the same criteria vector, only one of them is retained, arbitrarily. Let $MinPerm$ be a function which takes a set of job permutations as input and returns its Pareto Permutations set. $OptPerm(S)$ is the Pareto Permutations of all jobs from the jobset S and $OptPerm_l(S)$, $l \in \{1, ..., |OptPerm(S)|\}$ is the l-th permutation (the numeration is arbitrary).

Definition 2. *For a given job schedule in which all operations are executed as early as possible, its Critical Path (see for instance Brucker (2007); Kelley Jr and Walker (1959))*[1] *CP^j, $j \in \{1, 2, 3\}$, is defined as the sequence of operations on the first j machines, that are executed consecutively without any idle time and the total processing time of these operations determines C_j^M. More specifically, for any two consecutive operations in CP^j, denoted by $O_{i'j'} \rightarrow O_{i''j''}$, $i', i'' \in \{1, ..., n\}$, $j', j'' \in \{1, ..., j\}$, we must have $j' = j''$ or otherwise $i' = i''$ and $j'' = j' + 1$. In the latter case, job i' is defined as a critical job. For remaining jobs, let CP_q^j, $q \in \{1, ..., j\}$, be the set of jobs whose operations on machine q appear in CP^j, excluding critical jobs.*

As an example, in Figure 2.3, CP^2 consists of the sequence $(O_{11}, O_{21}, O_{31}, O_{32}, O_{42})$, with job 3 as the critical job. $CP_1^2 = \{1, 2\}$ and $CP_2^2 = \{4\}$. It should be noticed that for a given j, CP^j may not be unique, in that case we choose it arbitrarily.

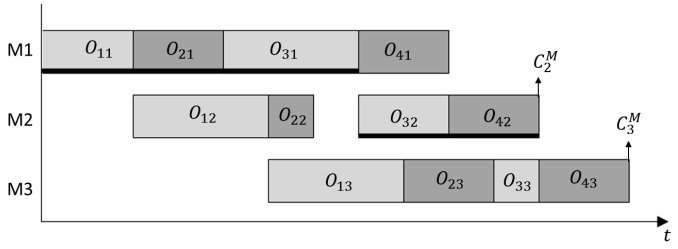


Figure 2.3: Example of *critical paths*

Lemma 1. *For a jobset S' with $|S'| = t$, the number of its Pareto Permutations $|OptPerm(S')|$ is at most $\mathcal{O}^*(2^t)$.*

Proof. Each permutation in $OptPerm(S')$ has a corresponding criteria vector $\langle C_2^M, C_3^M \rangle$. The number of *Pareto Permutations* corresponds to the number of non-dominated vectors, which is bounded by the number of possible values of C_2^M. The next observation is that the C_2^M value of a given permutation is related to the critical path of job operations on the first two machines (Definition 2). Hence the problem reduces to count the number of possible critical paths. To do this, we first decide the critical job, for which we have t choices. Each of the remaining jobs has to belong to either CP_1^2 or CP_2^2. This yields 2^{t-1} choices. Hence $|OptPerm(S')| \leq t2^{t-1} = \mathcal{O}^*(2^t)$. □

We now state in Lemma 2 another upper bound on $|OptPerm(S')|$.

[1] Instead of defining *Critical Path* in the context of digraph, here we adapt the definition to make it more intuitive

Lemma 2. *If there exists a constant M such that $\forall i, j$, $p_{ij} \leq M$, then for a given jobset S' of t jobs, the number of non-dominated criteria vectors $\langle C_2^M, C_3^M \rangle$, in Pareto Permutations, is at most $(t+1)M = \mathcal{O}^*(M)$.*

Proof. Lemma 2 is based on the consideration that the maximum value of C_2^M will not exceed $(t+1)M$. Since C_2^M is an integer, the number of possible values of C_2^M is also bounded by $(t+1)M$. □

The recurrence function of our dynamic programming algorithm is stated in Theorem 2.

Theorem 2. *$OptPerm(S)$ can be computed by Dynamic Programming as follows:*

$$OptPerm(S) = MinPerm\{OptPerm_l(S \setminus \{k\}).\{k\} : k \in S, l \in \{1, ..., |OptPerm(S \setminus \{k\})|\}\}$$

Theorem 2 is the recursive formula used by the *Dynamic Programming* algorithm. It is directly based on Theorem 1. When constructing an optimal solution starting with partial permutations, only *Pareto Permutations* are kept. Consider that we already have $OptPerm(S \setminus \{k\})$, $\forall k \in S$. In order to construct $OptPerm(S)$, we append job k after each permutation from $OptPerm(S \setminus \{k\})$ to get a new set of permutations for jobset S. Over all of these newly constructed permutations, we apply the operation $MinPerm$ to remove dominated ones and return $OptPerm(S)$. A more intuitive representation is given in Figure 2.4 where $|S| = t$.

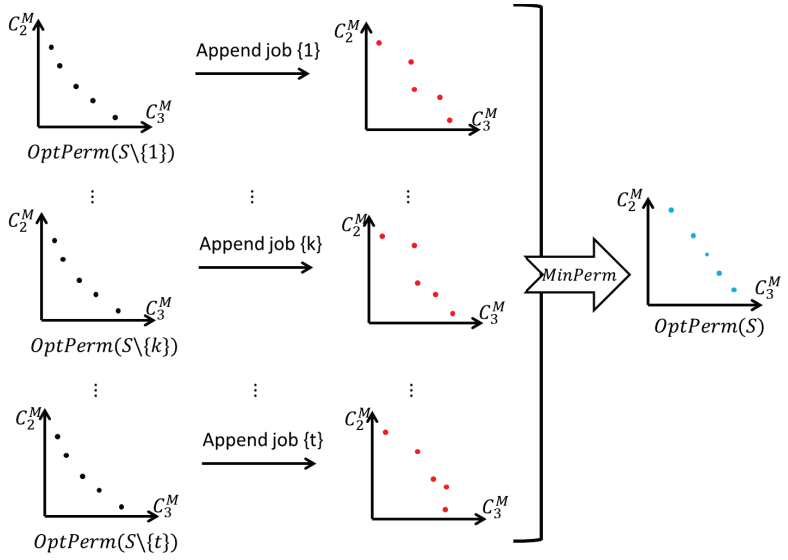


Figure 2.4: Recursive procedure of the *Dynamic Programming* algorithm.

2.3.1 Complexity Analysis

Consider implementing the algorithm in a recursive way according to Theorem 2. We keep in memory already computed $OptPerm(S)$, for all S, for fast access later. As initialization, when $S = \{j\}$, we set $OptPerm(S) = \{j\}$. From that point, every $OptPerm(S)$ for $|S| > 1$ can be computed by $MinPerm$ from $OptPerm(S \setminus \{k\})$, $\forall k \in S$. The function $MinPerm$ uses an existing algorithm (Kung et al., 1975) for finding, given a set of N vectors $\langle C_2^M, C_3^M \rangle$, non-dominated vectors, with a complexity of $\mathcal{O}(N \log N)$. According to the proof of Lemma 1, N is bounded by $t(t-1)2^{t-2} = \mathcal{O}(t^2 2^t)$ where the extra factor t is the number of possible values of k. Therefore, computing $OptPerm(S)$ from solved subproblems yields a complexity of

$$\mathcal{O}(t^2 2^t \log(t^2 2^t)) = \mathcal{O}(t^3 2^t) = \mathcal{O}^*(2^t)$$

The algorithm traverses across all problems defined by all subsets of jobs. Thus, the overall time complexity for calculating $OptPerm(S)$ is

$$\begin{aligned} \sum_{t=1}^{n} \binom{n}{t} \mathcal{O}(t^3 2^t) &= n^3 \sum_{t=1}^{n} \binom{n}{t} \mathcal{O}(2^t) 1^{n-t} \\ &= \mathcal{O}(n^3 (2+1)^n) && \text{(Binomial theorem)} \\ &= \mathcal{O}^*(3^n) \end{aligned}$$

Alternatively, based on Lemma 2, the time complexity can also be expressed as

$$\sum_{t=1}^{n} \binom{n}{t} \mathcal{O}^*(M) = \mathcal{O}^*(M 2^n)$$

The minimum value of C_{max}^M can be retrieved from $OptPerm(S)$ with an additive cost of $\mathcal{O}^*(2^n)$ (or $\mathcal{O}^*(M)$) time, which does not change the established complexity. The space complexity is also $\mathcal{O}^*(3^n)$ (or $\mathcal{O}^*(M2^n)$) considering the storage of all necessary *Pareto Permutations*. Therefore, we proved that at the cost of exponential memory usage the $F3||C_{max}$ problem can be solved in $\mathcal{O}^*(3^n)$ time. This improves the complexity result $2^{\mathcal{O}(n)} \times \|I\|^{\mathcal{O}(1)}$ of Jansen et al. (2013b).

2.3.2 Computational results

The practical efficiency of exact exponential algorithms is also an interesting subject that we want to explore in this thesis. These algorithms may not seem efficient in practice in general, because specific treatment is performed dedicated to obtain a theoretic guarantee on worst-case instances. However, despite this possible behavior of the proposed DP algorithm, we wanted to evaluate it in practice on randomly generated instances.Unfortunately, despite the superiority of our algorithm in worst-case scenarios with respect to existing exact algorithms, preliminary experimentation (Morin, 2014-2015) shows that the algorithm only managed to solve random instances with up to 20 jobs due to its space complexity.

N	5	10	15	20
Min time (s)	0	0	0.8	42.4
Max time (s)	0	0	2.8	82.5
Mean time (s)	0	0	1.2	54.1

Table 2.1: Preliminary results of dynamic programming

The results are reported in Table 2.1. The processing time of operations are generated uniformly from the range $[1, 100]$. For each problem size, 20 instances are generated. Small instances with 5 and 10 jobs are solved instantly. Instances with 15 and 20 jobs are also solved within an acceptable time, but the exponential growth of solution time over the size of instances can already be observed. On instances with 25 jobs, the program encountered "out of memory" problems on certain instances, knowing that the tests have been performed on a computer having 4G RAM and a 1.70GHz CPU.

This is far less than that can be solved by the *Branch & Bound* algorithm of Ladhari and Haouari (2005). However, it is of theoretical interest to have an algorithm with a proved running time upper bound faster than the brute force approach.

2.4 Generalizations to other flowshop problems

The *Dynamic Programming* algorithm can be generalized to other three-machine flowshop problems in a similar manner. In this section we present such generalizations on the $F3\|f_{max}$ and the $F3\|\sum f_i$ problems, with a dedicated complexity analysis provided.

2.4.1 The $F3\|f_{max}$ problem

The *Dynamic Programming* algorithm can be applied to a more general problem referred to as $F3\|f_{max}$. This problem is similar to the $F3||C_{max}$ problem but with a more general objective function f_{max} to be minimized. For job $i \in \{1, \ldots, n\}$, let C_i be the completion time of job i on machine 3 and let $f_i(C_i)$ be the cost associated to job i. We consider the f_i's as non-decreasing functions. The objective is defined as

$$f_{max} = max\{f_i(C_i)|i = 1, \ldots, n\}$$

If $f_i = f_j \ \forall i \neq j$, then the *Dynamic Programming* algorithm can be applied without adaptation, and therefore with the same $\mathcal{O}^*(3^n)$ worst-case complexity. The $F3||C_{max}$ problem can be considered as a particular case of this setting where $f(C_i) = C_i$.

If the definition of the function f_i depends on i, for instance $f_i = w_i C_i$ with w_i the weight of job i, then the dominance status of a given permutation is not only dependent on $\langle C_2^M, C_3^M \rangle$, but also on the corresponding f_{max} value of the permutation. In other words, given two different permutations π and π' for a same subset of jobs, the condition $C_2^M(\pi) < C_2^M(\pi')$ and $C_3^M(\pi) < C_3^M(\pi')$, is not sufficient to discard π' since it is possible that $f_{max}(\pi') < f_{max}(\pi)$ which means that π' may lead to an optimal solution. Therefore, we add f_{max} as a third criterion into the criteria vector. The *Dynamic Programming*

acts in a similar way as the one for $F3||C_{max}$ but on a three-dimension criteria space $\langle C_2^M, C_3^M, f_{max}\rangle$.

Theorem 3. *Let S be a set of n jobs to be scheduled, and $S' \subset S$ a subset of S. Given π and π' as two permutations of jobs from S', σ as a permutation of jobs from $S \setminus S'$, then we have:*

$$If \begin{cases} C_2^M(\pi) \leq C_2^M(\pi') \\ C_3^M(\pi) \leq C_3^M(\pi') \\ f_{max}(\pi) \leq f_{max}(\pi') \end{cases} \quad then \begin{cases} C_2^M(\pi.\sigma) \leq C_2^M(\pi'.\sigma) \\ C_3^M(\pi.\sigma) \leq C_3^M(\pi'.\sigma) \\ f_{max}(\pi.\sigma) \leq f_{max}(\pi'.\sigma) \end{cases}$$

that is, the partial permutation π dominates π'. Every schedule starting by π' is dominated by a schedule starting by π.

Proof. The proof of Theorem 3 is straightforward. □

The same running time analysis can be performed as before, where the critical part is to count the number of possible non-dominated vectors. This number is bounded by the number of different $\langle C_2^M, C_3^M\rangle$ values. Similarly to what has been done in Lemma 1 to establish an upper bound on the number of values of C_2^M, we can notice that there also exists a critical path (but through three machines, see Figure 2.5) which leads to each value of C_3^M and therefore the number of possible values of C_3^M is upper bounded by $\mathcal{O}^*(3^n)$. If we consider the number of possible $\langle C_2^M, C_3^M\rangle$ vectors as the number of combinations of C_2^M values and C_3^M values, then we have $2^n \times 3^n = 6^n$ non-dominated criteria vectors $\langle C_2^M, C_3^M\rangle$. However it can be proved that this upper bound is far from being tight, by exploiting links between CP^2 and CP^3.

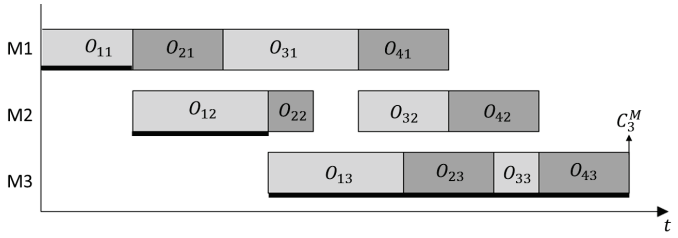
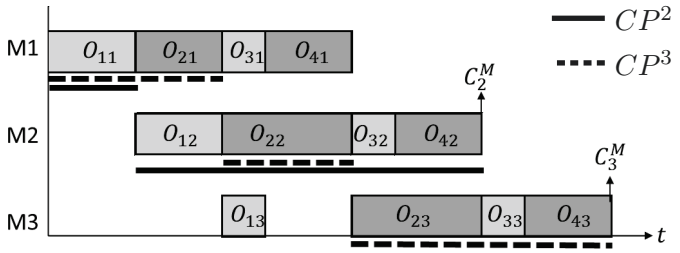


Figure 2.5: Example of a critical path through 3 machines

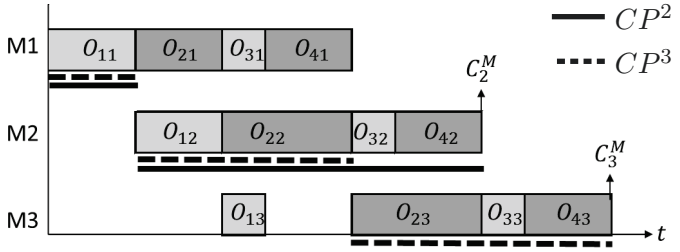
Lemma 3. *On machine 1, it is sufficient to consider critical paths such that $CP_1^3 \subseteq CP_1^2$.*

Proof. Firstly it is obvious that on machine 1, either $CP_1^3 \subseteq CP_1^2$ or $CP_1^2 \subseteq CP_1^3$, since either they are empty sets or they must contain the first several operations on machine 1. Consider the situation where we have $CP_1^2 \subsetneq CP_1^3$, $CP_2^3 \subsetneq CP_2^2$ holds since CP_2^2 contains all jobs of $N \setminus CP_1^2$ except the critical job. Now by setting $CP_1^3 = CP_1^2$ and adjust CP_2^3 accordingly, CP^3 remains a critical path and $CP_1^3 \subseteq CP_1^2$. An example is illustrated in Figure 2.6. □

Lemma 4. *On machine 2, only two cases need to be considered:*



(a) $CP_1^2 \subsetneq CP_1^3$



(b) Change the choice of CP^3

Figure 2.6: An example for Lemma 3. CP^2 (resp. CP^3) is indicated by a solid (resp. dashed) line.

1. $CP_1^3 = CP_1^2$ *and* $CP_2^3 \subseteq CP_2^2$.
2. $CP_1^3 \subsetneq CP_1^2$ *and* $CP_2^3 \cap CP_2^2 = \emptyset$.

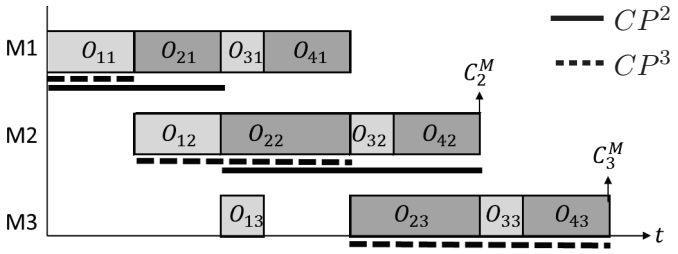
Proof. Both cases are easy to verify with a similar consideration as in Lemma 3. The first case is obvious, since $CP_1^3 = CP_1^2$ and CP_2^2 contains all jobs of $N \setminus CP_1^2$ except the critical job, which is also the critical job of CP^3, so $CP_2^3 \subseteq CP_2^2$ holds. In the second case $CP_1^3 \subsetneq CP_1^2$, if $CP_2^3 \cap CP_2^2 \neq \emptyset$, let job $o \in CP_2^3 \cap CP_2^2$. This means that both critical paths CP^2 and CP^3 contain the operation O_{o2}, and the operation sequences preceding O_{o2} are different in CP^2 and CP^3. Then similarly to Lemma 3, CP^2 or CP^3 can then be rearranged by adopting the same operation sequences preceding O_{o2} to reduce to the first case of Lemma 4. See Figure 2.7 and Figure 2.8 for an example. Note that we are only interested in the values of C_2^M and C_3^M, the actual critical paths are not our concern. □

Lemma 5. *For a jobset S' with $|S'| = t$, the number of possible pairs $\langle C_2^M, C_3^M \rangle$ is bounded by $\mathcal{O}^*(4^t)$.*

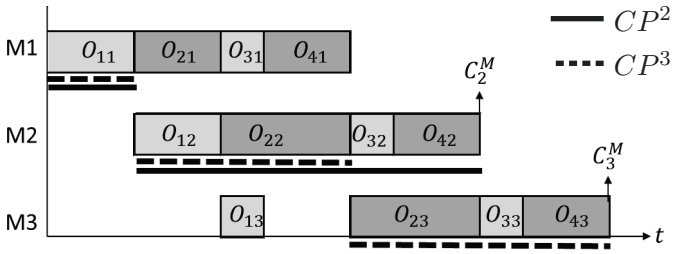
Proof. In order to count this number, we first need to choose the critical jobs for CP^2 and CP^3, as similar to the proof of Lemma 3. This yields a factor of $t(t-1) = \mathcal{O}(t^2)$ such jobs.

According to Lemma 4, we then choose i jobs that form CP_1^2 (CP_2^2 is then implicitly decided). Then, for CP^3:

- Consider Lemma 4 case 1: the i jobs are also in CP_1^3, the remaining $(t-i)$ jobs can be put in CP_2^3 or CP_3^3, which yields 2^{t-i} choices.



(a) $CP_1^3 \subsetneq CP_1^2$ and $CP_2^3 \cap CP_2^2 \neq \emptyset$



(b) Change the choice of CP^2

Figure 2.7: An example for Lemma 4. CP^2 (resp. CP^3) is indicated by a solid (resp. dashed) line.

- Consider Lemma 4 case 2 (see Figure 2.8): these i jobs can be put either in CP_1^3, CP_2^3 or CP_3^3. The remaining $t-i$ jobs can only be put in CP_3^3, since $CP_2^3 \cap CP_2^2 = \emptyset$. In total this yields 3^i choices.

$$\begin{aligned} t(t-1)\sum_{i=1}^{t}\binom{t}{i}\mathcal{O}(3^i + 2^{t-i}) &= \mathcal{O}(t^2(4^t + 3^t)) \\ &= \mathcal{O}(t^2 4^t) \\ &= \mathcal{O}^*(4^t) \end{aligned}$$

□

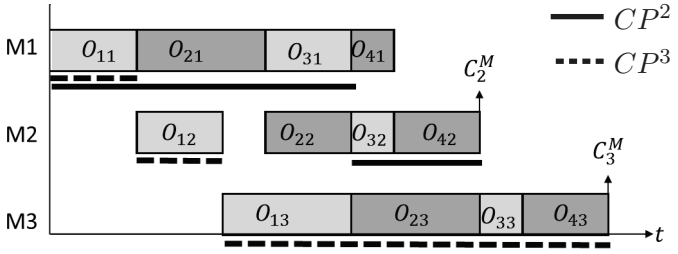


Figure 2.8: An example for Lemma 4, case 2. CP^2 (resp. CP^3) is indicated by a solid (resp. dashed) line.

Finally, with a same analysis as for the $F3||C_{max}$ problem, the complexity of our algorithm for the $F3\|f_{max}$ problem is:

$$\sum_{t=1}^{n} \binom{n}{t} \mathcal{O}(t \cdot t^2 4^t \log(t \cdot t^2 4^t)) = \mathcal{O}(t^4 4^n) = \mathcal{O}^*(5^n)$$

As in Lemma 2, we can also adopt a parameter M such that $p_{ij} \leq M$ into the analysis. In this case the number of possible pairs $\langle C_2^M, C_3^M \rangle$ is bounded by $(n+1)M \cdot (n+2)M = \mathcal{O}^*(M^2)$, and the final complexity is

$$\sum_{t=1}^{n} \binom{n}{t} M^2 = \mathcal{O}^*(M^2 2^n)$$

2.4.2 The $F3\| \sum f_i$ problem

Another problem that our *Dynamic Programming* can be applied to is $F3\| \sum f_i$, in which the objective is to minimize the $\sum f_i(C_i)$ problem. When we consider f_i as a non-decreasing function, this problem can be solved by *Dynamic Programming* in a similar way to the $F3\|f_{max}$ problem. Hence, based on a similar dominance condition between partial permutations, the criteria vector is defined as $\langle C_2^M, C_3^M, F \rangle$ with $F = \sum f_i(C_i)$ for a given sequence.

Theorem 4. *Let S be a set of n jobs to be scheduled, and $S' \subset S$ a subset of S. Given π and π' as two permutations of jobs from S', σ as a permutation of jobs from $S \setminus S'$, then we have:*

$$If \begin{cases} C_2^M(\pi) \leq C_2^M(\pi') \\ C_3^M(\pi) \leq C_3^M(\pi') \\ F(\pi) \leq F(\pi') \end{cases} then \begin{cases} C_2^M(\pi.\sigma) \leq C_2^M(\pi'.\sigma) \\ C_3^M(\pi.\sigma) \leq C_3^M(\pi'.\sigma) \\ F(\pi.\sigma) \leq F(\pi'.\sigma) \end{cases}$$

that is, the partial permutation π dominates π', and there exists a permutation schedule starting by π at least as good for the $\sum f_i$ as another starting by π'.

Proof. The proof of Theorem 4 is straightforward. □

Therefore with the same analysis as for the $F3\|f_{max}$ problem, the total complexity is also $\mathcal{O}^*(5^n)$ (or $\mathcal{O}^*(M^2 2^n)$) in time and space.

2.5 A general framework and its applications

From the applications of *Dynamic Programming* on the above flowshop problems, it can be observed that the algorithm is not limited to flowshop problems, but can be applied on any other problems verifying similar dominance conditions over *Pareto Front*. For this reason, we further formulate the idea as a general framework, then apply it onto several other scheduling problems.

We call *conventional DP* the classic algorithmic pattern of *Dynamic Programming* when applied to sequencing problems including TSP in 1960's (Bellman, 1962; Held and Karp,

1962). This pattern is also referred to as *Dynamic Programming across the subsets* according to Woeginger (2003). In fact the generalization of the conventional DP has been studied in the literature (Carraway and Morin, 1988; Mitten, 1974). Briefly speaking, the idea of generalized DP (GDP) is still to construct optimal solutions by optimal solutions of subproblems. It eliminates partial solutions that are not promising to be extended to an optimal global solution. Nevertheless, instead of using directly the global cost function to define the local dominance condition, as done in the conventional DP, GDP defines the dominance condition by problem-specific local preference relations. Preference is given to the decision(s) that can lead to a better global solution by considering all possible subsequent decisions. GDP has been widely discussed on multi-criteria optimization problems or problems with multi-attribute objective functions (see for instance Carraway et al. (1990); Villarreal and Karwan (1981); Evans et al. (1982); Bard et al. (1988)). The local preference rule in GDP is often based on the relation of criteria or attribute vectors associated to each partial solution and this vector is often explicit for the aforementioned problems.

Here we propose another formulation of generalized DP (named PDP, Pareto Dynamic Programming) which shares the same high-level principle as GDP but its formulation has a special focus on single criterion optimisation problems with implicit dominance vectors and on minimizing the computational complexity of the algorithm in worst-case scenarios. The general idea is to identify a vector of criteria which affect the objective function value so that partial decision sequences whose criteria vector is not in the corresponding Pareto sets can be removed during the process of DP. Generating Pareto sets in DP has already been mentioned in the literature (Henig, 1983; Rosenman and Gero, 1983) and it is sometimes considered as impractical due to the size of Pareto sets stored during the process. However, in the context of the design of *Exact Exponential Algorithms* for $\mathcal{NP}$-hard problems, PDP may provide a good performance guarantee in worst-case scenarios.

We first formulate PDP and discuss the complexity analysis of PDP in a general way. Then we apply it on some single machine scheduling problems with dynamic arrivals, for which the instantiation of PDP yields the best time complexity so far. Finally, we provide perspectives for its applications.

2.5.1 Pareto Dynamic Programming

Now let us formulate PDP in a general way, considering a general optimization problem where we need to find the optimal order of decisions to make. Let $\vec{x} \in \mathbb{R}^p$ be a criteria vector referring to the current state of solution, i.e. the consequence of decisions that have already been made. The state is initiated as a all-zero vector $\vec{x_0}$ when no decision has been made. Let $\vec{y} \in \mathbb{R}^q$ be the vector representing the next decision to make. Making the next decision can be represented by a binary operator of concatenation $\odot$ defined as follows:

$$\vec{x} \odot \vec{y} = \vec{x'},\ \vec{x'} \in \mathbb{R}^p$$

In addition, for $A \subset \mathbb{R}^p, \vec{y} \in \mathbb{R}^q$,

$$A \odot \vec{y} = \{\vec{x} \odot \vec{y} | \vec{x} \in A\}$$

That is, $A \odot \vec{y}$ is the set of solution states obtained by concatenating the decision $\vec{y}$ at the end of each decision state from A.

For $\vec{x}, \vec{x'} \in \mathbb{R}^p$, the local preference between $\vec{x}$ and $\vec{x'}$ is defined as

$$\vec{x} \preccurlyeq \vec{x'} \Rightarrow (\vec{x} \odot \vec{y}) \leq (\vec{x'} \odot \vec{y}), \forall \vec{y} \in \mathbb{R}^q$$

The relation $\preccurlyeq$ is read as "is preferred with respect to", considering a minimization problem. Ties are broken arbitrarily.

Define $S = \{\vec{y_1}, \ldots, \vec{y_n}\} \subset \mathbb{R}^q$ the candidate decisions at the input for a problem of size n. $Opt(S)$ is the set of final solution states containing the optimal ones, PDP can then be expressed by the following equation:

$$Opt(S) = \mathcal{M}in \left(\bigcup_{\vec{y} \in S} Opt(S \setminus \vec{y}) \odot \vec{y} \right) \tag{2.1}$$

with $\mathcal{M}in(\cdot)$ returning all non-dominated vectors from the input ones.

The correctness of equation 2.1 is obvious since the quality of a solution state can be judged directly on its associate state vector $\vec{x} \in \mathbb{R}^p$ by definition. The final optimal solution can be retrieved easily from $Opt(S)$ by evaluating the global objective function.

The definition of the criteria vector $\vec{x} \in \mathbb{R}^p$ is problem dependent and it plays a key role in order to apply PDP on a specific problem. For each decision permutation π, its associated criteria vector $\mathcal{V}(\pi) = \langle g_1(\pi), \ldots, g_p(\pi) \rangle$ should be defined in a way such that for any permutation π' of the same set of decisions as π, $\pi \preccurlyeq \pi'$ iff $\mathcal{V}(\pi) \leq \mathcal{V}(\pi')$. By finding (or defining) appropriate criterion g_i, we transfer our focus from decision sequences to a specific criteria space. In this context, $\mathcal{M}in$ is implemented to find the permutations whose associated criteria vector is minimum and $Opt(S)$ contains these permutations.

The problems that are treatable by the conventional DP (the one solving TSP, for instance) can be seen as a special case of PDP where $\mathcal{V} = \langle f \rangle$, i.e. the global criterion is the only one that needs to be considered for the local dominance. In this case $p = 1$ so that the cardinality of $Opt(S)$ is always 1.

2.5.2 Complexity analysis

In terms of the time complexity of PDP, it is easy to see that the main operation in the algorithm is the function $\mathcal{M}in$. For $|S| = t$, the number of input of $\mathcal{M}in$ in equation (2.1) is upper-bounded by $t \times |Opt(S \setminus \vec{y})|$. Therefore, it is important to bound the maximum cardinality of $Opt(S)$ for a given decision set S. According to the authors' experience, the upper-bound analysis of $Opt(S)$ is usually problem dependent and it is the place where we can exploit largely the properties of the problem. In section 2.5.3, we show the application of PDP on several problems together with detailed problem-dependent complexity analysis.

Suppose for $|S| = t$ the cardinality of $Opt(S)$ can be bounded by $B(t)$ which is a problem dependent function. The running time of function $\mathcal{M}in$ depends on p, which is the number of criteria in $\mathcal{V}$. According to Kung et al. (1975), $\mathcal{M}in$ runs in $\mathcal{O}(N \log N)$ for $p = 2, 3$ and in $\mathcal{O}(N(\log N)^{p-2})$ for $p \geq 4$, with N the number of input of $\mathcal{M}in$, i.e. $tB(t-1)$. Also considering the algorithm runs across all the subsets of different cardinality t, the total running time can then be computed as:

$$T(n) = \begin{cases} \sum_{t=1}^{n} \binom{n}{t} t \cdot B(t-1) = \sum_{t=1}^{n} \binom{n}{t} t \leq 2^n & p = 1 \quad (2.2) \\ \sum_{t=1}^{n} \binom{n}{t} t \cdot B(t-1) \log(B(t-1)) & p = 2, 3 \quad (2.3) \\ \sum_{t=1}^{n} \binom{n}{t} t \cdot B(t-1) (\log(B(t-1)))^{p-2} & p \geq 4 \quad (2.4) \end{cases}$$

Notice that for the case of equation (2.2), $B(t) = 1$, which explains the frequent occurrence of the complexity $\mathcal{O}(n2^n)$ of DP algorithms for many problems.

2.5.3 Illustration for the $1|r_i|\sum f_i(C_i)$ and $1|r_i|\max(f_i(C_i))$ problems

Consider a single machine scheduling problem with dynamic arrivals. Let $\{1, \ldots, n\}$ be the jobs to schedule, and each job i is defined by its processing time p_i and a release date r_i. For a given schedule, i.e. a sequence of the n jobs with every job starting as early as possible after its release date, C_i is defined as the completion time of job i. Let f_i be a non-decreasing function of C_i, and the problem (denoted to as $1|r_i|\sum f_i(C_i)$) asks to find an optimal schedule of jobs which minimizes $\sum f_i(C_i)$. This problem is $\mathcal{NP}$-hard even when $f_i = C_i$. Note that the conventional DP cannot be applied directly since the objective function $\sum f_i(C_i)$ alone is not sufficient to decide the dominance between two partial solutions composed by same jobs.

To apply PDP on the $1|r_i|\sum f_i(C_i)$ problem, we define $p = 2$ with $g_1(\pi)$ the completion time of a partial schedule π and $g_2(\pi) = \sum_{i \in \pi} f_i(C_i)$ its associated cost function. This is sufficient since if $\mathcal{V}(\pi) \leq \mathcal{V}(\pi')$, whatever is the job sequence after, the global solution starting with π always dominates π'. On the other hand, the vector $\vec{y_i} \in \mathbb{R}^q$ associated to decision i (job i, in our case) is pretty straightforward: $\vec{y_i} = \langle p_i, r_i \rangle$. That is, a job sequence $\pi = (2, 1, 3)$ can be seen as the consequence of making three decisions by doing $\vec{x_0} \odot \vec{y_2} \odot \vec{y_1} \odot \vec{y_3}$, with $\vec{x_0} \in \mathbb{R}^p$ the initial state vector containing only 0's. Moreover, $\mathcal{V}(\pi) \odot \vec{y_i} = \langle \max(g_1(\pi), r_i) + p_i, g_2(\pi) + f_i(\max(g_1(\pi), r_i) + p_i) \rangle$. Equation 2.1 can now be applied directly.

We now provide the complexity analysis of PDP on the $1|r_i|\sum f_i(C_i)$ problem. The question to answer first is, for a given jobset S, $|S| = t$, what is the value of $B(t)$, i.e. the maximum cardinality of $\{\mathcal{V}(\pi) | (\pi, \pi' \text{ are permutations of S}) \wedge (\nexists \pi',\ \mathcal{V}(\pi') < \mathcal{V}(\pi))\}$. A simple observation is that $B(t)$ is upper-bounded by the number of possible values of $g_1(\pi)$, which is bounded by the number of completion times of π. In order to compute this, we introduce Lemma 6.

Lemma 6. *Assume $r_i \geq 0$, $\forall i = 1, .., t$. Let $\pi = (j_1, j_2, \ldots, j_t)$ be a permutation of t jobs. Then, $\exists k \in \{1, \ldots, t\}$ such that $g_1(\pi) = r_k + \sum_{h=k}^{t} p_h$.*

Proof. Starting from the end of π, find the longest sub-sequence of jobs which do not have idle time in-between. Let k be the first job of this sub-sequence. If k is the first job in π, it

must start at time r_k since all jobs start as early as possible. Otherwise if k is not the first in π, we know it still starts at time r_k since there is some idle time between job j_{k-1} and j_k by the construction of the sub-sequence. An exemple of π is given in Figure 2.9. □

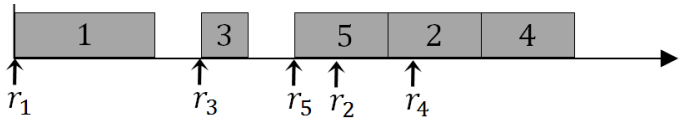


Figure 2.9: Example for Lemma 6 (k=5)

Lemma 7. $B(t) = \mathcal{O}(t2^{t-1}) = \mathcal{O}^*(2^t)$.

Proof. By Lemma6, possible values of $g_1(\pi)$ can be computed as follows. First choose k, for which there are t choices; then choose the jobs that are put after j_k, which yields $\mathcal{O}(2^{t-1})$ possibilities. □

Theorem 5. *Problem* $1|r_i|\sum f_i(C_i)$ *can be solved in* $\mathcal{O}^*(3^n)$ *time and space.*

Proof. The result can be derived directly from equation 2.3 and Lemma7:

$$T(n) = \sum_{t=1}^{n} \binom{n}{t} t \cdot \mathcal{O}^*(2^t) \log(\mathcal{O}^*(2^t)) = \mathcal{O}^*(n^2 \sum_{t=1}^{n} \binom{n}{t} 2^t \cdot 1^{n-t})) = \mathcal{O}^*(3^n)$$

□

The reasoning and the result is the same for the $1|r_i|\max(f_i(C_i))$ problem.

2.6 Auxiliary Result: complexity lower bound of $F3||C_{max}$ based on ETH

The conjecture $\mathcal{P} \neq \mathcal{NP}$ implies that $\mathcal{NP}$-hard problems cannot be solved in polynomial time. However, no more precision is given on the complexity of $\mathcal{NP}$-hard problems except that they are super-polynomial. People have realized that this simple binary classification of decision problems to *easy* or *hard* is perhaps too coarse-grained. This encourages to analyze the complexity of problems in a finer way. On this direction, Impagliazzo and Paturi (2001) proposed the *Exponential Time Hypothesis* (ETH) which states that 3SAT cannot be solved in sub-exponential time ($\mathcal{O}(2^{o(n)})$). Based on this assumption, complexity lower bounds have been deduced for many problems such as k-Colorability, k-Set Cover, Independent Set, Clique, and Vertex Cover. The existence of sub-exponential algorithms for any of them implies the same for the others. Moreover, a *Strong Exponential Time Hypothesis* (SETH) has also been proposed, which assumes that k-CNF-SAT requires $\mathcal{O}(2^n)$ time in the worst-case when k grows to infinity. This allows to derive even tighter bounds such as shown by Cygan et al. (2016): for every $\epsilon < 1$ an $\mathcal{O}(2^{\epsilon n})$ algorithm for Hitting Set, Set Splitting or NAE-SAT would violate SETH.

Being not able to propose an algorithm with a smaller upper bound on the worst-case time complexity than the proposed dynamic programming, we prove by ETH an asymptotic lower bound, by a reduction from 3SAT to SubsetSum and then to $F3||C_{max}$. We show that $F3||C_{max}$ cannot be solved in sub-exponential time, unless ETH fails. This immediately implies that the $Fm\|C_{max}$ problem, $\forall m > 3$, cannot be solved in sub-exponential time either.

When establishing this result, we were not aware that a similar result reported by Jansen et al. (2013b), stated that for $f \in \{C_{max}, \sum w_j C_j\}$, the problems $O3||f$, $J3||f$, $F3||f$ and $P2||f$ cannot be solved asymptotically faster than $(2^{O(n)} \cdot \|I\|)$ unless ETH fails. The used reduction chain 3SAT $\rightarrow$ SubsetSum$\rightarrow$ Partition $\rightarrow$ $F3||C_{max}$, which is different from ours.

2.6.0.1 From 3SAT to $F3||C_{max}$

As stated by Impagliazzo and Paturi (2001), proving lower bounds based on ETH implies designing problem reductions that preserve sub-exponential time property. For this, it is critical that resulting problem size of the reduction does not grow more than linearly. However, the time needed by the reduction is less important: it is not necessarily polynomial but should be at most sub-exponential.

Definition 3. *Consider a reduction $\propto$ from problem P_1 with complexity parameter m_1 to P_2 with complexity parameter m_2. Reduction $\propto$ is **sub-exponential time preserving** if*

1. *$\propto$ runs in sub-exponential time, i.e. for any $\epsilon > 0$, $\propto$ runs in $\mathcal{O}(2^{\epsilon n})$ and*
2. *$m_2(\propto(x)) = \mathcal{O}(m_1(x))$, i.e. the size of the output problem is linearly related to the size of the input problem.*

In fact, if P_2 can be solved in sub-exponential time, so can P_1, since for any given instance of P_1 we can construct in sub-exponential time an instance of P_2 that can be solved in sub-exponential time.

Many existing standard problem reductions are naturally sub-exponential preserving. It is pretty straightforward to verify this since standard reductions are in polynomial time and we only need to check whether "$m_2(\propto(x)) = \mathcal{O}(m_1(x))$". In the case of the $F3||C_{max}$ problem, it was proved as $\mathcal{NP}$-complete by Garey et al. (1976) by a reduction from 3-Partition (3PAR). The latter was proved to be $\mathcal{NP}$-complete by a series of reductions : 3SAT $\rightarrow$ 3-Dimentional Matchining (3DM) $\rightarrow$ 3PAR (Garey and Johnson, 1975). The classic reference of Garey and Johnson (1979) also described the reduction 3SAT $\rightarrow$ 3DM $\rightarrow$ 4PAR $\rightarrow$ 3PAR.

Unfortunately, the described reduction chain from 3SAT to $F3||C_{max}$ does not preserve sub-exponential time, since the reduction to 3DM outputs an instance of size $\Omega(n^2)$, i.e. asymptotically lower bounded by n^2. In this section, we describe a new reduction chain from 3SAT to $F3||C_{max}$ which is proved to be sub-exponential time preserving and which in result allows to prove lower bounds for $F3||C_{max}$ based on ETH. This chain is $3SAT \rightarrow$SubsetSum $\rightarrow F3||C_{max}$.

2.6.0.2 3SAT→SubsetSum

For this step, the standard NP-completeness reduction can be adopted directly (see for instance Cormen et al. (2009)). However, in order to prove that the resulting instance has its size linear in that of the input instance, we need to apply here the Sparsification Lemma of Impagliazzo and Paturi (2001). For the sake of clarity we start by a brief description of the reduction (R1). The desired input and output of R1 are described as follows:

Input problem (3SAT): Given a boolean formula F in 3-CNF form containing n variables and m clauses and each clause having exactly 3 literals, the question is whether there exists an assignment of variables that makes $F = true$.

Output problem (SubsetSum): Given S as a set of n integers, t an integer parameter, the question is whether there exists a subset $S' \subset S$, s.t. $\sum_{i \in S'} i = t$.

R1 constructs the target SubsetSum as follows. Each element in S is represented by a $(m+n)$-digit base-7 number. The first n digits correspond to variables and the last m digits are labeled by clauses. We note $C(i) = \{c : (c \text{ is a clause in } F) \wedge (v_i \in c)\}$. Without loss of generality, we assume that each variable must appear in at least one clause and no clause contains a variable and its negation.

1. The target value t is a $(m+n)$-digit base-7 number which has 1 on all variable digits and 4 on all clause digits.

2. For each variable v_i in F, we add 2 numbers into S: s_i and s'_i with the v_i-digit set to 1. $\forall c \in C(i)$, the c-digit of s_i (resp. s'_i) is set to 1 if c is satisfied when $v_i = true$ (resp. $false$). The other digits are set to 0.

3. For each clause c_i in F, we add 2 numbers a_i and a'_i into S. a_i (resp. a'_i) is created by setting the c_i-digit to 1 (resp. 2), and by setting the other digits to 0. Note that these are slack numbers which will help us to add to 4 in the target digit.

Clearly, in the digits of t, the $v_i - digit$ is 1 if and only if when s_i or s'_i is chosen, which corresponds to a true or false assignment to v_i. The $c_i - digit$ is 4 only when at least one variable assignment satisfies c_i since otherwise we would have at most 3 by adding up a_i and a'_i. Also note that by encoding the numbers in base 7, no carries can occur. The correctness of the reduction is therefore straightforward.

The resulting SubsetSum instance has $2(m+n)$ numbers, each has $(m+n)$ digits, the whole construction can be done in polynomial time (even when considering $m = n^3$). The size of the resulting instance is $2(m+n)$, but according to Definition 3, we should have $\mathcal{O}(n)$. This is the place where the Sparsification Lemma applies.

Lemma 8. Sparsification Lemma *(Impagliazzo and Paturi, 2001)*
$\forall \epsilon > 0, k \geq 2$, a k-SAT formula F with n variables and m clauses can be transformed into a k-SAT formula F' which is the disjunction of at most $2^{\epsilon n}$ sub-formulas, each with $\mathcal{O}(n)$ clauses. The transformation runs in $\mathcal{O}(poly(n)2^{\epsilon n})$ time. Alternatively speaking, k-SAT has a $2^{o(n)}$ algorithm if and only if it has a $2^{o(m)}$ algorithm.

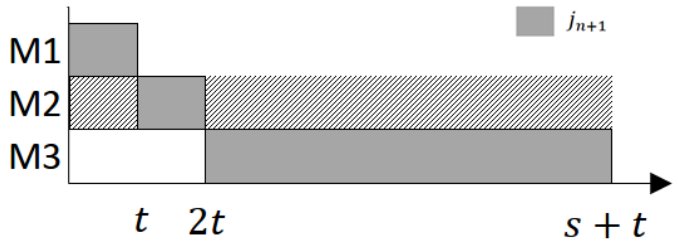


Figure 2.10: Scheduling overview of the $F3||C_{max}$ instance reduced from SubsetSum

This allows the reduction R1 to produce for each sub-formula with n' variables and m' clauses, a SubsetSum instance of size $2(m'+n') = \mathcal{O}(n')$. Therefore R1 is *sub-exponential time preserving* since if SubsetSum can be solved in sub-exponential time then for any instance of 3SAT and any $\epsilon > 0$, we can solve it in sub-exponential time by first applying the Sparsification Lemma then applying R1 and solve one by one the resulting SubsetSum sub-instances.

2.6.0.3 SubsetSum $\rightarrow F3||C_{max}$

Now we focus on the reduction (R2) from SubsetSum to $F3||C_{max}$. The desired input and output of R2 are described as follows:

Input problem (SubsetSum): Given S as a set of n integers, t an integer parameter, the question is whether there exists a subset $S' \subset S$, s.t. $\sum_{i \in S'} i = t$.

Output problem ($F3||C_{max}$): Given n' jobs to be processed on 3 machines, each job must be processed on machine $1, 2, 3$ in this order without overlapped processing. The processing time of job j on the 3 machines is denoted as $p_j = \langle p_{j1}, p_{j2}, p_{j3} \rangle$. Each machine can only handle one job at one time. We ask whether there exists an ordering of jobs such that the whole processing can finish before time instant T.

We note s as the sum of all integers in S. The reduction (R2) constructs the set of jobs J as follows.

1. Add to J job j_{n+1}: $\langle t, t, s-t \rangle$
2. For each element $a_i \in S$, add job i : $\langle 0, a_i, 0 \rangle$
3. Let $T = s + t$

Since the execution of j_{n+1} requires already $s + t$ time, as shown in Figure 2.10, the remaining n jobs must be perfectly filled into the line-shaded area without leaving any idle time in order to not excede T on machine 2. Since that area is cut into two parts of size t and $(s-t)$ respectively, a perfect fill-in of the first part implies the existence of $S' \subset S$ such that $\sum S' = t$.

Lemma 9. *Assuming ETH, $\nexists$ algorithm such that for any $\epsilon > 0$, it solves $F3||C_{max}$ in $\mathcal{O}(2^{\epsilon n})$ time. In other words, the $F3||C_{max}$ problem cannot be solved in sub-exponential time unless ETH fails.*

Proof. It is easy to check that R2 runs in linear time and the resulting $F3||C_{max}$ instance has size $n + 2$ which is nearly the same as the input instance. As a consequence, R2 is sub-exponential time preserving. Combining with R1, the lemma is proved. □

2.7 Chapter Summary

In this work we focus on the design of exact algorithms for several sequencing problems considering worst-case scenarios. The first problem that is tackled is the $F3||C_{max}$ problem for which we propose a *Dynamic Programming* algorithm, improving the best-known time complexity in the literature from $2^{O(n)} \times \|I\|^{\mathcal{O}(1)}$ (which can be proved to be $\mathcal{O}^*(c^n)$, $c > 3$, when applied to this problem) to $\mathcal{O}^*(3^n)$. The generalizations of the algorithm on the $F3\|f_{max}$ and $F3\|\sum f_i$ problems are then presented with worst-case time and space complexities in $\mathcal{O}^*(5^n)$ or $\mathcal{O}^*(M^2 2^n)$. Since the idea is based on the consideration of dominance conditions for non-decreasing cost functions and on the consideration of *Pareto Front* of corresponding criteria vectors, it can be easily generalized to other problems that have similar structures. We formulate the generalized algorithm as a framework named *Pareto Dynamic Programming*, and then apply it to the $1|r_i|\sum f_i(C_i)$ and $1|r_i|\max(f_i(C_i))$ problems. More generally, for all problems that possess a dominance condition based on multiple criteria, our *Dynamic Programming* is likely to be applicable. Table 2.2 summaries the complexity results of PDP when applied to several problems. Recall that $B(n)$ is the upper bound of the number of optimal solutions returned after solving an instance of size n.

Problem $(\alpha\|\beta\|f)$ $\beta \neq pmtn$ $f \in \{f_{max}, \sum f_i\}$	Criteria Vector	B(n)	Complexity
$1\|\|f$	$\langle f \rangle$	1	$\mathcal{O}^*(2^n)$
$1\|r_i\|f$	$\langle C_{max}, f \rangle$	$\mathcal{O}^*(2^n)$	$\mathcal{O}^*(3^n)$
$F2\|\|f$	$\langle C_{max}, f \rangle$	$\mathcal{O}^*(2^n)$	$\mathcal{O}^*(3^n)$
$F3\|\|f$	$\langle C_2^M, C_{max}, f \rangle$	$\mathcal{O}^*(4^n)$	$\mathcal{O}^*(5^n)$

Table 2.2: Application of PDP on sequencing problems

The main work in this chapter has been performed together with Christophe Lenté, Mathieu Liedloff and Vincent T'Kindt. The results on the $F3||C_{max}$, $F3\|f_{max}$ and $F3\|\sum f_i$ problems have been published in *Journal of Scheduling* (Shang et al., 2017c), *MISTA* 2015 conference (Shang et al., 2015) and *ROADEF* 2016 conference (Shang et al., 2016b).

Chapter 3

Branch & Merge on the Single Machine Total Tardiness Problem

3.1 Introduction

In this chapter, we report new results on exact exponential algorithms solving a pure sequencing problem, the single machine total tardiness problem, denoted by $1||\sum T_i$. In this problem, a job set $N = \{1, 2, \ldots, n\}$ of n jobs must be scheduled on a single machine. For each job i, a processing time p_i and a due date d_i are given. The problem asks for arranging the job set into a sequence σ so as to minimize $T(N, \sigma) = \sum_{i=1}^{n} T_i = \sum_{i=1}^{n} \max\{C_i - d_i, 0\}$, where C_i is the completion time of job j in sequence σ. The $1||\sum T_i$ problem is NP-hard in the ordinary sense (Du and Leung, 1990). It has been extensively studied in the literature and many exact procedures (Lawler, 1977; Potts and Van Wassenhove, 1982; Della Croce et al., 1998; Szwarc et al., 2001) have been proposed. The current state-of-the-art exact method of Szwarc et al. (2001) dates back to 2001 and solves to optimality instances with up to 500 jobs. All these procedures are search tree approaches, but dynamic programming algorithms were also considered. On the one hand, Lawler (1977) proposed a pseudo-polynomial dynamic programming algorithm running with complexity $\mathcal{O}(n^4 \sum p_i)$. On the other hand, the standard technique of doing dynamic programming across the subsets (see, for instance, Fomin and Kratsch (2010)) applies and runs with complexity $\mathcal{O}(n^2 2^n)$ both in time and space. Latest theoretical developments for the problem, including both exact and heuristic approaches can be found in the survey of Koulamas (2010).

We keep the use of the $\mathcal{O}^*(\cdot)$ notation for measuring worst-case complexities. $T(n)$ is still defined as the time required in the worst-case to exactly solve a problem of size n. To the best of our knowledge, no available exact algorithm for this problem running in $\mathcal{O}^*(c^n)$ (c being a constant) time and polynomial space has been reported in the literature. Actually, the complexity of the state-of-the-art algorithm of Szwarc et al. (2001) was not discussed by the authors, but our analysis shows that it runs in $\mathcal{O}^*(2.4143^n)$ time and polynomial space (see section 3.5.1). One could also possibly apply a divide-and-conquer approach as described by Gurevich and Shelah (1987) and Bodlaender et al. (2012). The idea is to determine the half of jobs that should be processed first by enumerating all the 2-partitions of the input jobset. Then, for each 2-partition, solve the two corresponding

subproblems recursively and finally combine the obtained solutions of subproblems to get the optimal solution. This would lead to an $\mathcal{O}^*(4^n)$ complexity in time requiring polynomial space. The aim of this work is to present an improved exact algorithm exploiting known decomposition properties of the problem. Different versions of the proposed approach are described in section 3.2. A final version making use of a new technique called *Branch & Merge* that avoids the solution of several equivalent subproblems in the branching tree is presented in section 3.3. Finally we present some additional results in section 3.5.

First, let use recall some known properties on the $1||\sum T_j$ problem.

Given the job set $N = \{1, 2, \ldots, n\}$, let $(1, 2, \ldots, n)$ be a LPT (Longest Processing Time first) sequence, where $i < j$ whenever $p_i > p_j$ (or $p_i = p_j$ and $d_i \leq d_j$). Let also $([1], [2], \ldots, [n])$ be an EDD (Earliest Due Date first) sequence, where $i < j$ whenever $d_{[i]} < d_{[j]}$ (or $d_{[i]} = d_{[j]}$ and $p_{[i]} \leq p_{[j]}$). As the cost function is a regular performance measure, we know that in an optimal solution, the jobs are processed with no interruption starting from time zero. Let B_i and A_i be the sets of jobs that precede and follow job i in an optimal sequence. Consequently, in this optimal sequence being constructed, $C_i = \sum_{\ell \in B_i} p_\ell + p_i$. Similarly, if job j is assigned to position k, we denote by $C_i(k)$ the corresponding completion time of i and by $B_i(k)$ and $A_i(k)$ the sets of predecessors and successors of i, respectively.

The main known theoretical properties are the following.

Property 1. *(Emmons, 1969) Consider two jobs i and j with $p_i < p_j$. Then, i precedes j in an optimal schedule if $d_i \leq \max\{d_j, C_j\}$. Else j precedes i in an optimal schedule if $d_i + p_i > C_j$.*

Property 2. *(Lawler, 1977) Let job 1 in LPT order correspond to job $[k]$ in EDD order. Then, job 1 can be set only in positions $h \geq k$ and the jobs preceding and following job 1 are uniquely determined as $B_1(h) = \{[1], [2], \ldots, [k-1], [k+1], \ldots, [!ht]\}$ and $A_1(h) = \{[h+1], \ldots, [n]\}$.*

Property 3. *(Lawler, 1977; Potts and Van Wassenhove, 1982; Szwarc, 1993) Consider $C_1(h)$, for $h \geq k$. Job 1 cannot be set in positions $h \geq k$ if:*

(a) *$C_1(h) \geq d_{[h+1]}$, $\forall h < n$;*

(b) *$C_1(h) < d_{[r]} + p_{[r]}$, for some $r = k+1, \ldots, h$.*

Property 4. *(Szwarc and Mukhopadhyay, 1996) For any pair of adjacent positions $(i, i+1)$ that can be assigned to job 1, at least one of them is eliminated by Property 3.*

In terms of complexity analysis, we recall (see, for instance, Eppstein (2001)) that, if it is possible to bound above $T(n)$ by a recurrence expression of the type $T(n) \leq \sum_{i=1}^{h} T(n - r_i) + \mathcal{O}(p(n))$, then we have $\sum_{i=1}^{h} T(n - r_i) + \mathcal{O}(p(n)) = \mathcal{O}^*(\alpha(r_1, \ldots, r_h)^n)$ where $\alpha(r_1, \ldots, r_h)$ is the largest root of the function $f(x) = 1 - \sum_{i=1}^{h} x^{-r_i}$. This observation will be useful later on to analyze worst-case complexities.

3.2 A *Branch & Reduce* approach

We first focus on two direct *Branch & Reduce* algorithms based on the structural properties stated in the previous section. *Branch & Reduce* algorithms, as introduced

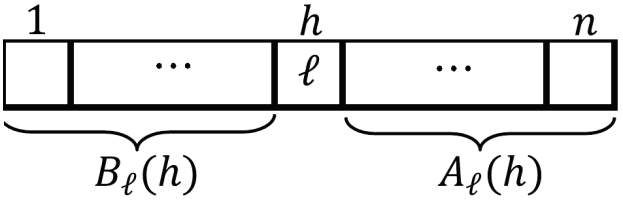


Figure 3.1: The situation when the longest job ℓ is put in position h

in section 1.2.2.1, are search tree based algorithms which are elaborated upon two main components: a branching strategy and a reduction rule. The later relies on a mathematical condition which enables to take optimal decisions in the worst-case scenario whenever some other decisions have been taken by the branching. For instance, it can be a rule which states that even in the worst-case, if we branch on a job j there always exists another job i which can be scheduled without branching on it.

3.2.1 A first *Branch & Reduce* algorithm

A basic *Branch & Reduce* algorithm TTBR1 (Total Tardiness *Branch & Reduce* version 1) can be designed by exploiting Property 2, which enables to decompose the problem into two smaller subproblems when the position of the longest job ℓ is given (see Figure 3.1).

The basic idea is to iteratively branch by assigning job ℓ to every eligible branching position and correspondingly decomposing the problem. Each time job ℓ is assigned to a certain position i, two different subproblems are generated, corresponding to schedule the jobs before ℓ (inducing subproblem $B_\ell(i)$) or after ℓ (inducing subproblem $A_\ell(i)$), respectively. The algorithm operates by applying to any given job set S starting at time t function $TTBR1(S,t)$ that computes the corresponding optimal solution. With this notation, the original problem is indicated by $N = \{1,...,n\}$ and the optimal solution is reached when function $TTBR1(N,0)$ is computed.

The algorithm proceeds by solving the subproblems along the branching tree according to a depth-first strategy and runs until all the leaves of the search tree have been reached. Finally, it provides the best solution found as an output. Algorithm 1 summarizes the structure of this approach, while Proposition 1 states its worst-case complexity. Figure 3.2 illustrates the branching operation of the algorithm.

Proposition 1. *Algorithm TTBR1 runs in $\mathcal{O}^*(3^n)$ time and polynomial space in the worst case.*

Proof. According to Property 2, the worst-case scenario arises when the ordering LPT is the same as EDD, in which case all positions are eligible for branching on the longest job. Whenever the longest job 1 is assigned to the first and the last position of the sequence, two subproblems of size $(n-1)$ are generated. For each $2 \leq i \leq n-1$, two subproblems with size $(i-1)$ and $(n-i)$ are generated. Hence, the total number of generated subproblems is $(2n-2)$ and the time cost related to computing the best solution of size n starting from these subproblems is $\mathcal{O}(p(n))$. This induces the following recurrence for the running time

Algorithm 1 Total Tardiness *Branch & Reduce* version 1 (TTBR1)

Input: $N = \{1, ..., n\}$ is the problem to be solved

1: **function** TTBR1(S, t)
2: $seqOpt \leftarrow$ the EDD sequence of jobs S
3: $\ell \leftarrow$ the longest job in S
4: **for** any eligible position i (Property 2) **do**
5: Branch by assigning job ℓ in position i
6: $seqLeft \leftarrow$ TTBR1$(B_\ell(i), t)$
7: $seqRight \leftarrow$ TTBR1$(A_\ell(i), t + \sum_{k \in B_\ell(i)} p_k + p_\ell)$
8: $seqCurrent \leftarrow$ concatenation of $seqLeft$, ℓ and $seqRight$
9: $seqOpt \leftarrow$ best solution between $seqOpt$ and $seqCurrent$
10: **end for**
11: **return** $seqOpt$
12: **end function**

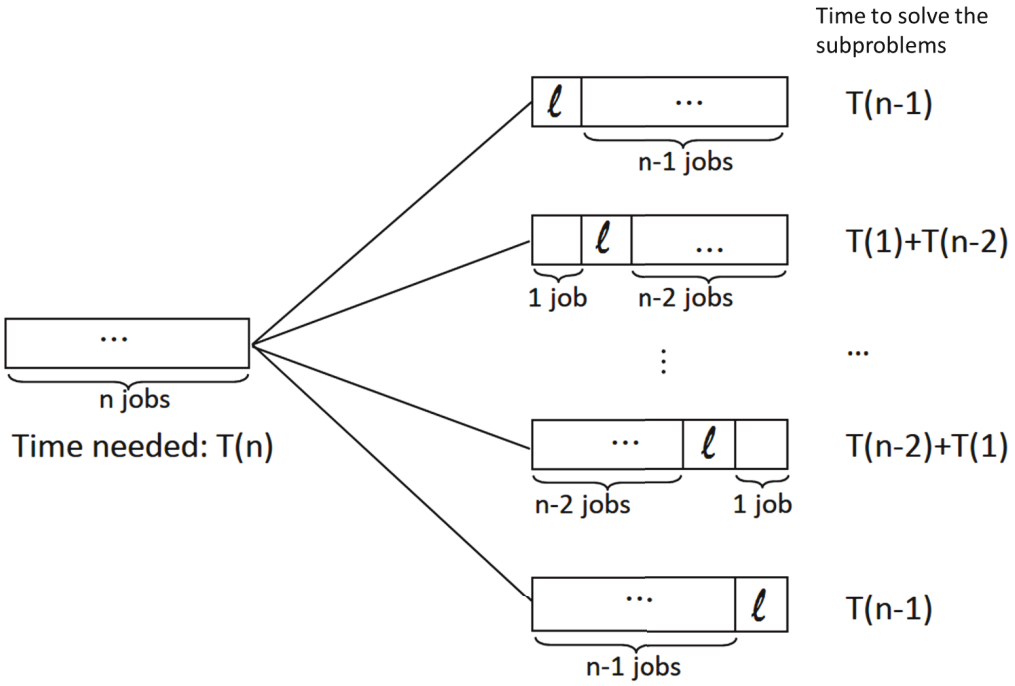


Figure 3.2: Branching scheme of TTBR1

$T(n)$ required by TTBR1:

$$T(n) = 2T(n-1) + 2T(n-2) + ... + 2T(2) + 2T(1) + \mathcal{O}(p(n)) \tag{3.1}$$

By replacing n with $(n-1)$, the following expression is derived:

$$T(n-1) = 2T(n-2) + ... + 2T(2) + 2T(1) + \mathcal{O}(p(n-1)) \tag{3.2}$$

Expression 3.2 can be used to simplify the right hand side of expression 3.1 leading to:

$$T(n) = 3T(n-1) + \mathcal{O}(p(n)). \tag{3.3}$$

By solving the corresponding equation $x^n = 3x^{n-1}$ and take its root as the exponential base, this recurrence yields the result $T(n) = \mathcal{O}^*(3^n)$. The space requirement is polynomial since the search tree is explored according to a depth-first strategy. □

3.2.2 A second *Branch & Reduce* algorithm

An improved version of the algorithm TTBR1 is defined by taking into account Property 3 and Property 4, which state that for each pair of adjacent positions $(i, i+1)$, at least one of them can be discarded. The worst case occurs when the largest possible subproblems are kept when branching, since otherwise the complexity can be easily proved as smaller. This corresponds to solving problems with size $(n-1), (n-3), (n-5), \ldots$, that arise by branching in positions i and $(n-i+1)$ with i odd. The resulting algorithm is referred to as TTBR2 (Total Tardiness Branch and Reduce version 2). Its structure is similar to the one of TTBR1 depicted in Algorithm 1, but lines 5-9 are executed only when ℓ can be set in position i according to Property 3. The complexity of the algorithm is discussed in Proposition 2.

Proposition 2. *Algorithm TTBR2 runs in* $\mathcal{O}^*((1+\sqrt{2})^n) = \mathcal{O}^*(2.4143^n)$ *time and polynomial space in the worst case.*

Proof. The proof is close to that of Proposition 1. We refer to problems where n is odd, but the analysis for n even is substantially the same. The algorithm induces a recursion of the type:

$$T(n) = 2T(n-1) + 2T(n-3) + ... + 2T(4) + 2T(2) + \mathcal{O}(p(n)) \tag{3.4}$$

as the worst case occurs when we keep the branches that induce the largest possible subproblems. Analogously to Proposition 1, we replace n by $(n-2)$ in the previous recurrence and we obtain:

$$T(n-2) = 2T(n-3) + 2T(n-5) + ... + 2T(4) + 2T(2) + \mathcal{O}(p(n-2)). \tag{3.5}$$

Again, we plug the latter expression into the former one and obtain the recurrence:

$$T(n) = 2T(n-1) + T(n-2) + \mathcal{O}(p(n)). \tag{3.6}$$

By solving the corresponding equation $x^n = 2x^{n-1} + x^{n-2}$ and take its largest positive root as the exponential base, the recurrence induces $T(n) = \mathcal{O}^*((1+\sqrt{2})^n) = \mathcal{O}^*(2.4143^n)$. The space complexity remains polynomial as for TTBR1. □

3.3 A *Branch & Merge* Algorithm

In this section, we describe how to get an algorithm running with complexity arbitrarily close to $\mathcal{O}^*(2^n)$ in time and polynomial space by integrating a node-merging procedure into TTBR1. The idea of such a procedure comes from the observation that in the search tree created by the branching scheme (Property 2), a lot of identical subproblems are explored for nothing.

We recall that in TTBR1 the branching scheme is defined by assigning the longest unscheduled job to each available position and accordingly divide the problem into two subproblems. To facilitate the description of the algorithm, we focus on the scenario where the LPT sequence $(1, ..., n)$ coincides with the EDD sequence $([1], ..., [n])$, for convenience we write $LPT = EDD$.

We provide the algorithmic details of the node-merging procedure on this scenario to facilitate its understanding. The resulting branch-and-merge algorithm has its time complexity tend to $\mathcal{O}^*(2^n)$. We prove by Lemma 15 that the case where $LPT = EDD$ is the worst-case scenario, hence, it follows that the problem $1||\sum T_i$ can be solved in time complexity tending to $\mathcal{O}^*(2^n)$. We leave to the reader the generalization of the node-merging procedure to the general case.

Figure 3.3 shows how an input problem $\{1, ..., n\}$ is decomposed by the branching scheme of TTBR1. Each node is labelled by the corresponding subproblem P_j (P denotes the input problem). Notice that from now on $P_{j_1,j_2,...,j_k}, 1 \leq k \leq n$, denotes the problem (corresponding to a node in the search tree) induced by the branching scheme of TTBR1 when the largest processing time job 1 is in position j_1, the second largest processing time job 2 is in position j_2 and so on till the k-th largest processing time job k being placed in position j_k. Notice that $P_{j_1,j_2,...,j_k}$ results from a series of branching operations according to Property 2, and may contain more than one subproblems to be solved. Consider the problem P_3 illustrated in Figure 3.3: both jobsets $\{2, 3\}$ and $\{4, .., n\}$ are to be scheduled and can be considered as two independents subproblems. Nevertheless, the notation $P_{j_1,j_2,...,j_k}$ can largely simplify the presentation of our algorithm and the fact that a node may contain more than one subproblems should not cause ambiguity to our belief. Surely, this is considered when analyzing the complexity of the algorithms.

To roughly illustrate the guiding idea of the merging technique introduced in this section, consider Figure 3.3. Noteworthy, nodes P_2 and $P_{1,2}$ are identical except for the initial subsequence (21 vs 12). This fact implies, in this particular case, that the problem of scheduling job set $\{3, ..., n\}$ at time $p_1 + p_2$ is solved twice. This kind of redundancy can however be eliminated by merging node P_2 with node $P_{1,2}$ and creating a single node

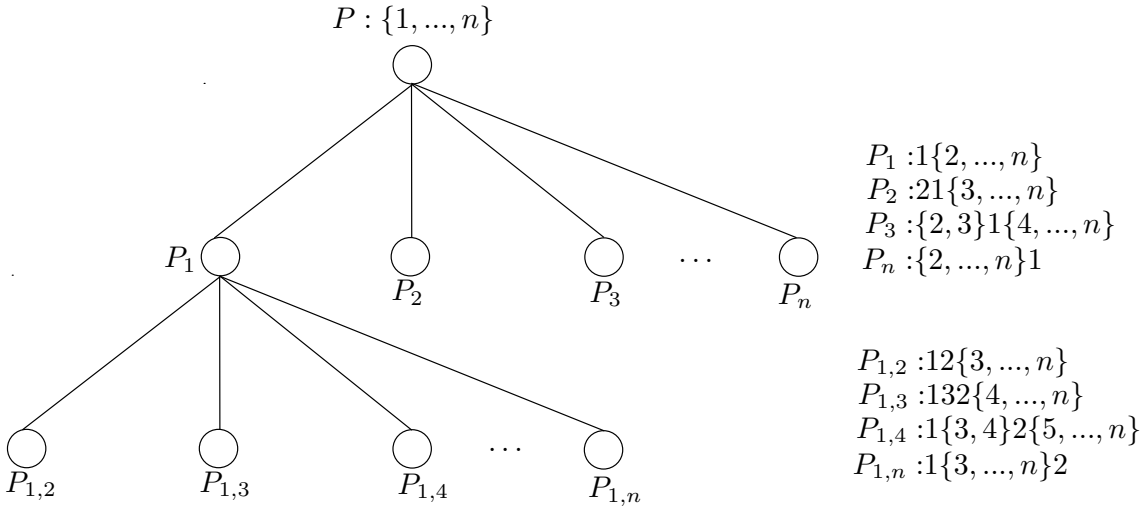


Figure 3.3: The branching scheme of TTBR1 at the root node

in which the best sequence among 21 and 12 is scheduled at the beginning and the job set $\{3, ..., n\}$, starting at time $p_1 + p_2$, remains to be branched on. Furthermore, the best subsequence (starting at time $t = 0$) between 21 and 12 can be computed in constant time. Hence, the node created after the merging operation involves a constant time preprocessing step plus the search for the optimal solution of job set $\{3, ..., n\}$ to be processed starting at time $p_1 + p_2$. We remark that, in the branching scheme of TTBR1, for any constant $k \geq 3$, the branches corresponding to P_i and P_{n-i+1}, with $i = 2, ..., k$, are decomposed into two problems where one subproblem has size $(n - i)$ and the other problem has size $(i - 1) \leq k$. Correspondingly, the merging technique presented on problems P_2 and $P_{1,2}$ can be generalized to all branches inducing subproblems of size less than k. Notice that, by means of algorithm TTBR2, any problem of size less than k requires at most $\mathcal{O}^*(2.4143^k)$ time to be solved (which is constant time when k is fixed). This is the central idea of the merging procedure: for a given value of k as an input of the algorithm, merge nodes which involve the same subproblems of size less than k. As k is a constant, the merging can be done in constant time.

In the remainder of the paper, for any constant $k \leq \frac{n}{2}$, we denote by left-side branches the search tree branches corresponding to problems $P_1, ..., P_k$ and by right-side branches the ones corresponding to problems $P_{n-k+1}, ..., P_n$.

In the following subsections, we show how the node-merging procedure can be systematically performed to improve the time complexity of TTBR1. Basically, two different recurrent structures hold respectively for left-side and right-side branches and allow to generate less subproblems at each level of the search tree. The node-merging mechanism is described by means of two distinct procedures, called `LEFT_MERGE` (applied to left-side branches) and `RIGHT_MERGE` (applied to right-side branches), which are discussed in sections 3.3.1 and 3.3.2, respectively. The final *Branch & Merge* algorithm is described in section 3.3.3 and embeds both procedures into the structure of TTBR1.

3.3.1 Merging left-side branches

We first illustrate the merging operations on the root node. The following proposition highlights two properties of problems P_j and $P_{1,j}$ with $2 \leq j \leq k$.

Lemma 10. *For a pair of problems P_j and $P_{1,j}$ with $2 \leq j \leq k$, the following conditions hold:*

1. *The solution of problems P_j and $P_{1,j}$ involves the solution of a common subproblem which consists in scheduling job set $\{j+1, ..., n\}$ starting at time $t = \sum_{i=1}^{j} p_i$.*
2. *Both in P_j and $P_{1,j}$, at most k jobs have to be scheduled before job set $\{j+1, ..., n\}$.*

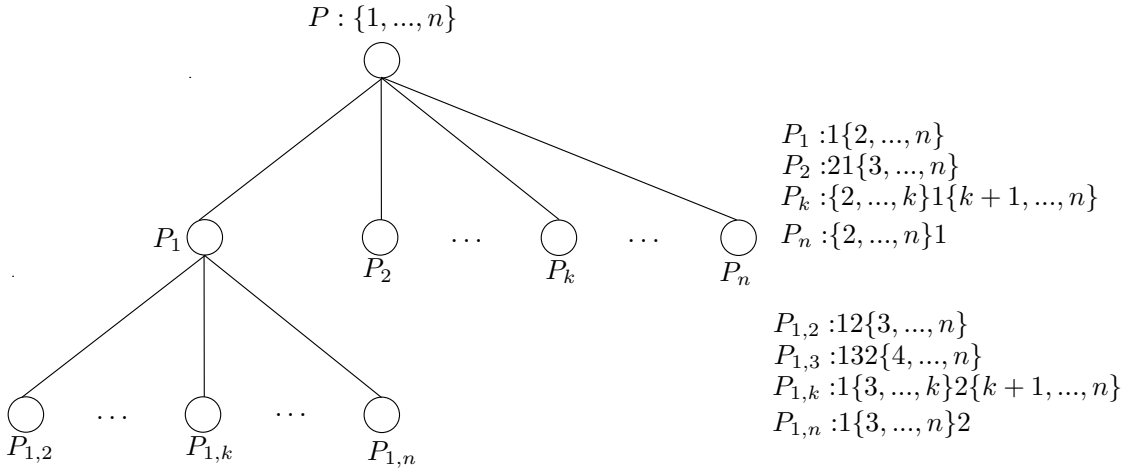
Proof. As problems P_j and $P_{1,j}$ are respectively defined by $\{2, ..., j\}1\{j+1, ..., n\}$ and $1\{3, ..., j\}2\{j+1, ..., n\}$, the first part of the property is straightforward.
The second part can be simply established by counting the number of jobs to be scheduled before job set $\{j+1, ..., n\}$ when j is maximal, *i.e.* when $j = k$. In this case, job set $\{k+1, ..., n\}$ has $(n-k)$ jobs which implies that k jobs remain to be scheduled before that job set. □

Each pair of problems indicated in Proposition 10 can be merged as long as they share the same subproblem to be solved. More precisely, $(k-1)$ problems P_j (with $2 \leq j \leq k$) can be merged with the corresponding problems $P_{1,j}$.

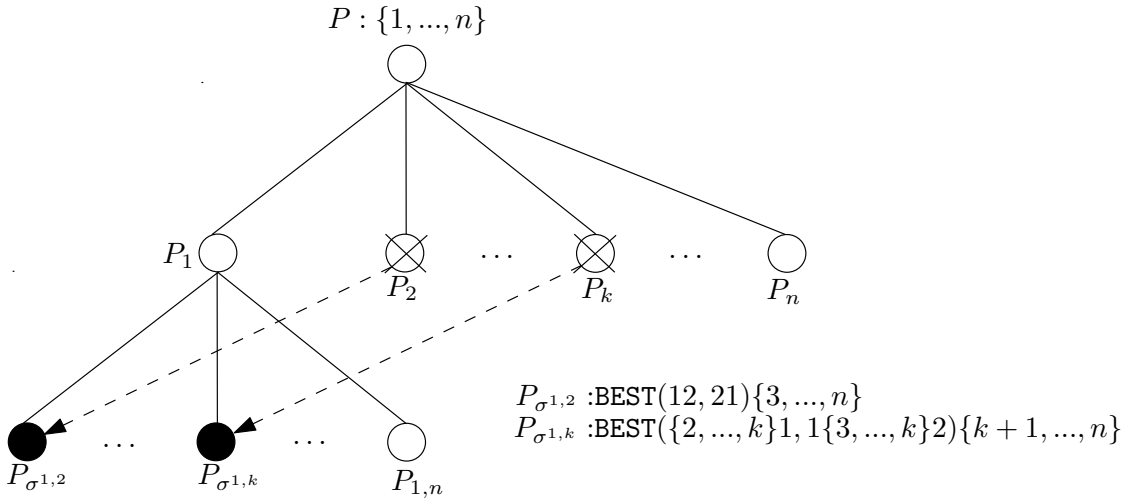
Figure 3.4 illustrates the merging operations performed at the root node on its left-side branches, by showing the branch tree before and after (Figure 3.4a and Figure 3.4b) such merging operations. For any given $2 \leq j \leq k$, problems P_j and $P_{1,j}$ share the same subproblem $\{j+1, ..., n\}$ starting at time $t = \sum_{i=1}^{j} p_i$. Hence, by merging the left part of both problems which is constituted by job set $\{1, ..., j\}$ having size $j \leq k$, we can delete node P_j and replace node $P_{1,j}$ in the search tree by the node $P_{\sigma^{1,j}}$ which is defined as follows (Figure 3.4b):

- $\{j+1, ..., n\}$ is the set of jobs on which it remains to branch.
- Let $\sigma^{1,j}$ be the sequence of branching positions on which the j longest jobs $1, ..., j$ are branched, that leads to the best jobs permutation between $\{2, ..., j\}1$ and $1\{3, ..., j\}2$. More concretely, the subproblem involving $\{2, ..., j\}$ from $\{2, ..., j\}1$ and the subproblem involving $\{3, ..., j\}$ from $1\{3, ..., j\}2$ must be solved (by calling TTBR2, for instance) first in order to obtain the optimal permutation of $\{2, ..., j\}1$ and $1\{3, ..., j\}2$. This involves the solution of two problems of size at most $(k-1)$ (in $\mathcal{O}^*(2.4143^k)$ time by TTBR2) and the comparison of the total tardiness value of the two sequences obtained.

In the following, we describe how to apply analogous merging operations on any node of the tree. With respect to the root node, the only additional consideration is that the children nodes of an arbitrary node may have already been affected by a previous merging. In Figure 3.4b, this is for instance the case for $P_{\sigma}^{1,3}$ which can be merged with $P_{\sigma^{1,2},3}$, a child node of $P_{\sigma}^{1,2}$ (if $k \geq 3$).



(a) Left-side branches of P before performing the merging operations



(b) Left-side branches of P after performing the merging operations.
BEST(α, β) returns the best sequence among α and β.

Figure 3.4: Left-side branches merging at the root node

In order to define the branching scheme used with the LEFT_MERGE procedure, a data structure $\mathcal{L}_\sigma$ is associated to a problem P_σ. It represents a list of $(k-1)$ subproblems that result from a previous merging and are now the first $(k-1)$ children nodes of P_σ. When P_σ is created by branching, $\mathcal{L}_\sigma = \emptyset$. When a merging operation sets the first $(k-1)$ children nodes of P_σ to $P_{\sigma^1}, ..., P_{\sigma^{k-1}}$, we set $\mathcal{L}_\sigma = \{P_{\sigma^1}, ..., P_{\sigma^{k-1}}\}$. This list if used to memorize the fact that the first $(k-1)$ child nodes of P_σ will not have to be built by a branching operation. As a conclusion, the following branching scheme for an arbitrary node of the tree holds.

Definition 4. *The branching scheme for an arbitrary node P_σ is defined as follows:*

- *If $\mathcal{L}_\sigma = \emptyset$, use the branching scheme of TTBR1;*

- *If $\mathcal{L}_\sigma \neq \emptyset$, extract problems from $\mathcal{L}_\sigma$ as the first $(k-1)$ branches, then branch on the longest job in P_σ in the available positions from the k-th to the last according to Property 2.*

This branching scheme, whenever necessary, will be referred to as ***improved branching****. Note that this will be extended later when "merging right-side nodes" is introduced.*

Before describing how merging operations can be applied on an arbitrary node P_σ, we highlight its structural properties by means of Proposition 3.

Proposition 3. *Let P_σ be a problem to branch on, and σ be the permutation of positions assigned to jobs $1, \ldots, |\sigma|$, with σ empty if no positions are assigned. The following properties hold:*

1. *$j^* = |\sigma| + 1$ is the job to branch on,*
2. *j^* can occupy in the branching process, positions $\{\ell_b, \ell_b + 1, \ldots, \ell_e\}$, where*

$$\ell_b = \begin{cases} |\sigma| + 1 & \text{if } \sigma \text{ is a permutation of } \{1, \ldots, |\sigma\} \text{ or } \sigma \text{ is empty} \\ \rho_1 + 1 & \text{otherwise} \end{cases}$$

with $\rho_1 = \max\{i : i > 0,$ positions $1, \ldots, i$ are in $\sigma\}$ and

$$\ell_e = \begin{cases} n & \text{if } \sigma \text{ is a permutation of } \{1, \ldots, |\sigma|\} \text{ or } \sigma \text{ is empty} \\ \rho_2 - 1 & \text{otherwise} \end{cases}$$

with $\rho_2 = \min\{i : \ i > \rho_1,\ i \in \sigma\}$

Proof. According to the definition of the notation P_σ, σ is a sequence of positions that are assigned to the longest $|\sigma|$ jobs. Since we always branch on the longest unscheduled job, the first part of the proposition is straightforward. The second part aims at specifying the range of positions that job j^* can occupy. Two cases are considered depending on the content of σ:

- If σ is a permutation of $\{1, \ldots, |\sigma|\}$, it means that the longest $|\sigma|$ jobs are set on the first $|\sigma|$ positions, which implies that the job j^* should be branched in positions $|\sigma| + 1$ to n
- If σ is not a permutation of $\{1, \ldots, |\sigma|\}$, it means that the longest $|\sigma|$ jobs are not set on consecutive positions. As a result, the current unassigned positions may be split into several ranges. As a consequence of the decomposition property and the assumption LPT=EDD, the longest job j^* should necessarily be branched on the first range of free positions, that goes from l_b to l_e. Let us consider as an example $P_{1,9,2,8}$ (see Figure 3.5), whose structure is $13\{5, \ldots, 9\}42\{10, \ldots, n\}$ and the job to branch on is 5. In this case, we have: $\sigma = (1, 9, 2, 8)$, $\ell_b = 3$, $\ell_e = 7$. It is easy to verify that 5 can only be branched in positions $\{3, \ldots, 7\}$ as a direct result of Property 2.

□

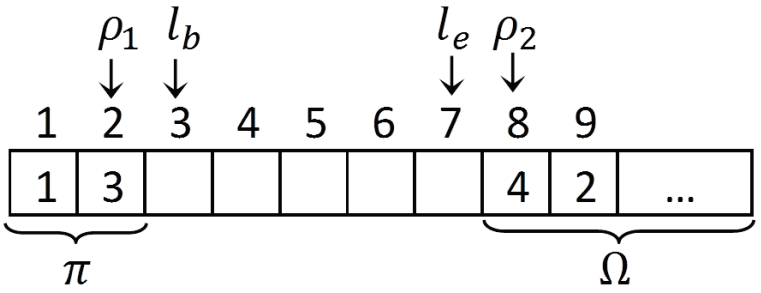


Figure 3.5: An example ($P_{1,9,2,8}$) for Proposition 3 and Corollary 1

Corollary 1 emphasises the fact that even though a node may contain several ranges of free positions, only the first range is the current focus since we only branch on the longest job in eligible positions.

Corollary 1. *Problem P_σ has the following structure:*

$$\pi\{j^*, \ldots, j^* + \ell_e - \ell_b\}\Omega$$

with π the subsequence of jobs on the first $(\ell_b - 1)$ positions in σ and Ω the remaining subset of jobs to be scheduled after position ℓ_e (some of them can have been already scheduled). The merging procedure is applied on job set $\{j^*, \ldots, j^* + \ell_e - \ell_b\}$ starting at time $t_\pi = \sum_{i \in \Pi} p_i$ where Π is the job set of π.

The validity of merging on a general node still holds as indicated in Proposition 4, which extends the result stated in Proposition 10.

Proposition 4. *Let P_σ be an arbitrary problem and let $\pi, j^*, \ell_b, \ell_e, \Omega$ be computed relatively to P_σ according to Corollary 1. If $\mathcal{L}_\sigma = \emptyset$ the j-th child node P_{σ^j} is P_{σ,ℓ_b+j-1} for $1 \leq j \leq k$. Otherwise, the j-th child node P_{σ^j} is extracted from $\mathcal{L}_\sigma$ for $1 \leq j \leq k-1$, while it is created as P_{σ,ℓ_b+k-1} for $j=k$. For any pair of problems P_{σ^j} and P_{σ^1,ℓ_b+j-1} (the $(j-1)$-th child node of P_{σ^1}) with $2 \leq j \leq k$, the following conditions hold:*

1. *Problems P_{σ^j} and P_{σ^1,ℓ_b+j-1} with $2 \leq j \leq k$ have the following structure:*

   - P_{σ^j}:

   $$\begin{cases} \pi^j\{j^*+j, \ldots, j^*+\ell_e-\ell_b\}\Omega & 1 \leq j \leq k-1 \text{ and } \mathcal{L}_\sigma \neq \emptyset \\ \pi\{j^*+1, \ldots, j^*+j-1\}j^*\{j^*+j, \ldots, j^*+\ell_e-\ell_b\}\Omega & (1 \leq j \leq k-1; \mathcal{L}_\sigma = \emptyset) \text{ or } j=k \end{cases}$$

   - P_{σ^1,ℓ_b+j-1}:
     $\pi^1\{j^*+2, \ldots, j^*+j-1\}(j^*+1)\{j^*+j, \ldots, j^*+\ell_e-\ell_b\}\Omega$

2. *By solving all the problems of size less than k, that consist in scheduling the job set* $\{j^*+1,\ldots,j^*+j-1\}$ *between* π *and* j^* *and in scheduling* $\{j^*+2,\ldots,j^*+j-1\}$ *between* π^1 *and* j^*+1*, both* P_{σ^j} *and* P_{σ^1,ℓ_b+j-1} *consist in scheduling* $\{j^*+j,...,j^*+\ell_e-\ell_b\}\Omega$ *starting at time* $t_{\pi^j}=\sum_{i\in\Pi^j} p_i$ *where* Π^j *is the job set of* π^j*.*

Proof. The first part of the statement follows directly from Definition 4 and simply defines the structure of the children nodes of P_σ. The problem P_{σ^j} is the result of a merging operation between a sibling node of P_σ and the problem P_{σ,ℓ_b+j-1} and it could possibly coincide with P_{σ,ℓ_b+j-1}, being the result of merging, for each $j=1,...,k-1$. Furthermore, P_{σ^j} is exactly P_{σ,ℓ_b+j-1} for $j=k$. The structure of P_{σ,ℓ_b+j-1} is $\pi\{j^*+1,\ldots,j^*+j-1\}j^*\{j^*+j,\ldots,j^*+\ell_e-\ell_b\}\Omega$, and the merging operations preserve the job set to schedule after j^*. Thus, we have $\Pi^j=\Pi\cup\{j^*,...,j^*+j-1\}$ for each $j=1,...,k-1$, and this proves the first statement. Analogously, the structure of P_{σ^1,ℓ_b+j-1} is $\pi^1\{j^*+2,\ldots,j^*+j-1\}(j^*+1)\{j^*+j,\ldots,j^*+\ell_e-\ell_b\}\Omega$. Once the subproblem before (j^*+1) of size less than k has been solved, P_{σ^1,ℓ_b+j-1} consists in scheduling the job set $\{j^*+j,...,j^*+\ell_e-\ell_b\}$ at time $t_{\pi^j}=\sum_{i\in\Pi^j} p_i$. In fact, we have that $\Pi^j=\Pi^1\cup\{j^*+2,\ldots,j^*+j-1\}\cup\{j^*+1\}=\Pi\cup\{j^*,\ldots,j^*+j-1\}$. Therefore, both P_{σ^j} and P_{σ^1,ℓ_b+j-1} consist in scheduling $\{j^*+j,...,j^*+\ell_e-\ell_b\}\Omega$ starting at time $t_{\pi^j}=\sum_{i\in\Pi^j} p_i$.

□

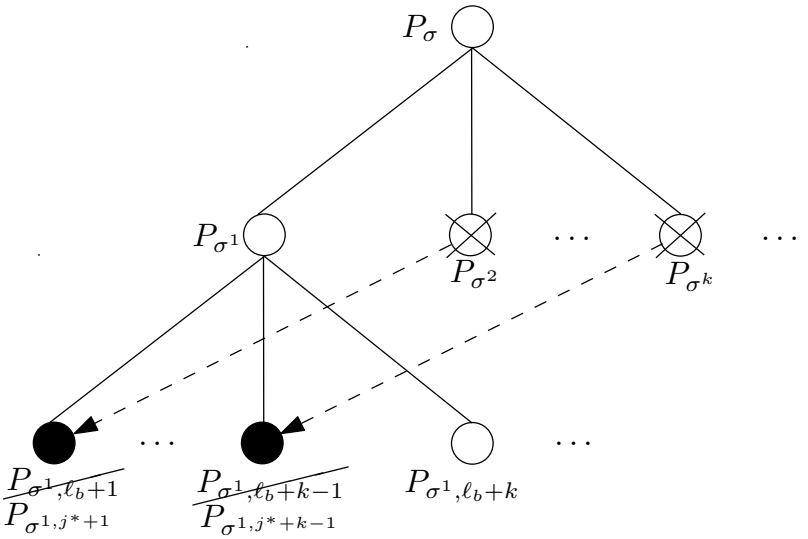


Figure 3.6: Merging for an arbitrary left-side branch

Analogously to the root node, each pair of problems indicated in Proposition 4 can be merged. Again, $(k-1)$ problems P_{σ^j} (with $2 \leq j \leq k$) can be merged with the corresponding problems P_{σ^1,ℓ_b+j-1}. P_{σ^j} is deleted and P_{σ^1,ℓ_b+j-1} is replaced by $P_{\sigma^{1,j^*+j-1}}$ (Figure 3.6), defined as follows:

- $\{j^*+j,...,j^*+\ell_e-\ell_b\}\Omega$ is the set of jobs on which it remains to branch on.
- Let σ^{1,j^*+j-1} be the sequence of positions on which the (j^*+j-1) longest jobs $\{1,...,j^*+j-1\}$ are branched, that leads to the best jobs permutation between π^j

and $\pi^1\{j^*+2,\ldots,j^*+j-1\}(j^*+1)$, for $2 \leq j \leq k-1$, and between $\pi\{j^*+1,\ldots,j^*+j-1\}j^*$ and $\pi^1\{j^*+2,\ldots,j^*+j-1\}(j^*+1)$, for $j=k$. This involves the solution of one or two problems of size at most $(k-1)$ (in $\mathcal{O}^*(2.4143^k)$ time by TTBR2) and the finding of the sequence that has the smallest total tardiness value knowing that both sequences start at time 0.

The `LEFT_MERGE` procedure is presented in Algorithm 2. Notice that, from a technical point of view, this algorithm takes as input one problem and produces as an output its first child node to branch on, which replaces all its k left-side children nodes.

Algorithm 2 `LEFT_MERGE` Procedure

Input: P_σ an input problem of size n, with ℓ_b, j^* accordingly computed
Output: Q: a list of problems to branch on after merging

1: **function** `LEFT_MERGE`(P_σ)
2: $Q \leftarrow \emptyset$
3: **for** j=1 to k **do**
4: Create P_{σ^j} (j-th child of P_σ) by the improved branching with the subproblem induced by the job set $\{j^*+1,\ldots,j^*+j-1\}$ solved if $\mathcal{L}_\sigma=\emptyset$ or $j=k$
5: **end for**
6: **for** j=1 to $k-1$ **do**
7: Create $P_{\sigma^1 j}$ (j-th child of P_{σ^1}) by the improved branching with the subproblem induced by the job set $\{j^*+2,\ldots,j^*+j-1\}$ solved if $\mathcal{L}_{\sigma^1}=\emptyset$ or $j=k$
8: **end for**
9: $\mathcal{L}_{\sigma^1} \leftarrow \emptyset$
10: **for** j=1 to $k-1$ **do**
11: $\mathcal{L}_{\sigma^1} \leftarrow \mathcal{L}_{\sigma^1} \cup$ `BEST`$(P_{\sigma^{j+1}}, P_{\sigma^1 j})$
12: **end for**
13: $Q \leftarrow Q \cup P_{\sigma^1}$
14: **return** Q
15: **end function**

Lemma 11. *The* `LEFT_MERGE` *procedure returns one node to branch on in* $\mathcal{O}(n)$ *time and polynomial space. The corresponding problem is of size* $(n-1)$.

Proof. The creation of problems P_{σ^1,ℓ_b+j-1}, $\forall j=2,\ldots,k$, can be done in $\mathcal{O}(n)$ time. The call of TTBR2 costs constant time. The `BEST` function called at line 8 consists in computing then comparing the total tardiness value of two known sequences of jobs starting at the same time instant: it runs in $\mathcal{O}(n)$ time. The overall time complexity of `LEFT_MERGE` procedure is then bounded by $\mathcal{O}(n)$ time as k is a constant. Finally, as only node P_{σ^1} is returned, its size is clearly $(n-1)$ when P_σ has size n. □

In the final part of this section, we discuss the extension of the algorithm in the case where $LPT \neq EDD$. In this case, Property 2 allows to discard subproblems associated to branching in some positions. Notice that if a problem P can be discarded according to this property, then we say that P does not exist and its associated node is empty.

Lemma 12. *When left merging is incorporated into the branching scheme of TTBR, instances such that $LPT = EDD$ correspond to worst-case instances.*

Proof. We prove the result by showing that the time reduction obtained from left merging and Property 2 in the case LPT=EDD is not greater than that of any other cases. Let us consider the improved branching scheme. The following exhaustive cases hold:

1. $1 = [1]$ and $2 = [2]$;
2. $1 = [j]$ with $j \geq 2$;
3. $1 = [1]$ and $2 = [j]$ with $j \geq 3$.

We first sketch the idea of the proof. For each of the 3 cases above, we analyse the time reduction that can be obtained on one single branching and merging and we show that the reduction corresponding to case 1, which covers the case LPT=EDD, is the smallest among all the 3 cases. In fact, for case 2 and case 3, some nodes are not created due to Property 2, and the resulting time reduction is not less than that of case 1.

Let $T(n)$ be the time needed to solve an instance of size n in general. From Lemma 2, we can deduce that $T(n) > 2T(n-1)$ because for instances with $LPT = EDD$, on each branching, a node of size $(n-1)$ is returned by left merging and another node of size $(n-1)$ exists due to the last child node of P_σ. This statement is also valid in the worst case if no merging is done, due to the branching scheme. The inequality $T(n) > 2T(n-1)$ induces that $T(n) = \omega(2^n)$, which will be used below to prove the lemma.

In order to be general, consider the current node as P_σ, as shown in Figure 3.6. The time reduction of the 3 cases are denoted respectively by $TR1$, $TR2$ and $TR3$. We also note $TR_{LPT=EDD}$ the time reduction corresponding to the case $LPT = EDD$.

In case 1, no nodes are eliminated by Property 2, hence, the merging can be done as described for the case LPT=EDD (Figure 3.6). Therefore, $TR1 = TR_{LPT=EDD} = T(n-2) + T(n-3) + .. + T(n-k)$ according to Lemma 11 when `LEFT_MERGE` is executed.

In case 2, the subproblem of P_σ corresponding to branching the longest job on the first position, is eliminated directly by Property 2. Therefore, $TR2 \geq T(n-1)$.

In case 3, let ℓ_b be the first free position in P_σ, as defined in Proposition 3. Some child nodes of P_{σ^1}, as in Figure 3.6, corresponding to branch job 2 on positions $\{\ell_b + 1, .., \ell_b + j - 1\}$, are eliminated due to Property 2. For these nodes, the time reduction that could have been achieved by merging is already ensured, while the nodes that are not eliminated, notably those corresponding to branch job 2 in positions $\{\ell_b + j, .., \ell_b + k - 1\}$ can still be merged pairwise with nodes $\{P_{\sigma^{j+1}}, .., P_{\sigma^k}\}$. More reduction can be gained if $j > k$. Therefore, $TR3 \geq T(n-2) + T(n-3) + .. + T(n-k)$.

Since $TR1 \leq TR3$, this brings us to compare $TR1$ and $TR2$. Suppose $TR1 > TR2$, i.e., $T(n-1) < T(n-2) + T(n-3) + .. + T(n-k)$, then we have $T(n-1) < T(n-2) + T(n-3) + .. + T(1)$. By solving this recurrence relation we get $T(n) = o(2^n)$ which is in contradiction with the fact that $T(n) = \omega(2^n)$, as proved above. Therefore, $TR1 < TR2$, i.e., on each recursion of the algorithm, the time reduction obtained in case 1 is not greater than any other cases. Since $TR_1 = TR_{LPT=EDD}$, this proves that $LPT = EDD$ is the

worst-case scenario, in which the `LEFT_MERGE` procedure returns one node of size $n-1$ to branch on. □

3.3.1.1 A working example for Left Merge

In order to better illustrate the merging operations on left-side branches, a detailed example is provided in this section. The input data is in Table 3.1 and the value of k is chosen as 2.

i	1	2	3	4
p_i	7	4	2	1
d_i	2	4	6	8

Table 3.1: A sample instance

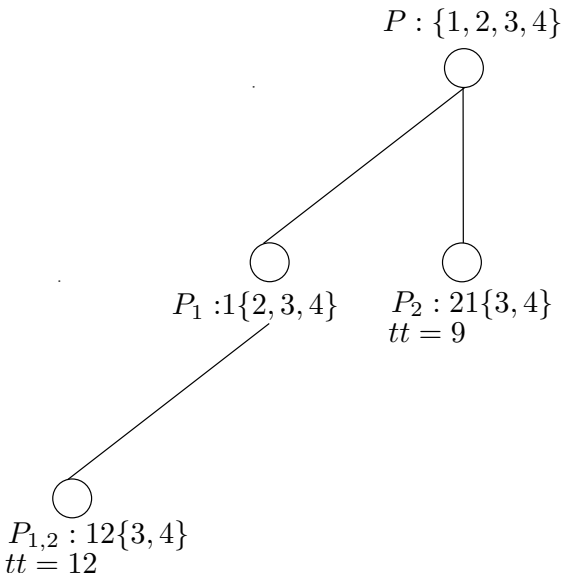
The solution of the instance is depicted in Figures 3.7a, 3.7b,3.7c,3.7d,3.7e. On each node, the jobs to schedule are surrounded by accolades and the indicated partial total tardiness value ($tt = \sum T_j$) is computed on the jobs that are fixed before the unscheduled jobs. The applied algorithm is TTBR1 but with `LEFT_MERGE` integrated. The algorithm runs in depth-first. `LEFT_MERGE` is firstly called on the root node P. Lines 1-8 in Algorithm 2 results the situation in Figure 3.7a. This firstly (lines 3-5) involves the generation of the first k child nodes of P which are P_1 and P_2, with $\mathcal{L} = \{P_1\}$, and the computation of total tardiness of the fixed job sequences. For instance, the sequence $(2,1)$ in P_2 has a total tardiness of 9. Similarly, by lines 6-8, the first $(k-1)$ child nodes of P_1 are generated and the total tardiness of partially fixed jobs is also computed. The sequence $(1,2)$ in $P_{1,2}$ has a total tardiness of 12. Then, by comparing the total tardiness of the fixed parts of P_2 and $P_{1,2}$, line 11 merges node P_2 to the position of $P_{1,2}$ by setting the child list of P_1 $\mathcal{L}_1 = \{P_2\}$, since $9 < 11$ (see Figure 3.7b). The result of merging is P_2 which is renamed as $P_{\sigma^{1,2}}$ according to our notation. Line 14 returns the next node to open, which is P_1. This completes the first call of `LEFT_MERGE`.

Then the algorithm continues in the same way by applying `LEFT_MERGE` on the node P_1. In a similar way, the node $P_{1,3}$ and the first child node of $P_{\sigma^{1,2}}$ are generated and their partial total tardiness values are computed. By comparing the partial total tardiness of these two nodes, $P_{1,3}$ is cut since $17 > 16$ (Figure 3.7c). This also explains why we decide to *merge* two nodes instead of just cutting the dominated one. Actually, by putting the merged node in a specific position, the node can further participate in subsequent merging operations. More concretely, if P_2 was not moved to the position of $P_{1,2}$, the merging in Figure 3.7c would not happen.

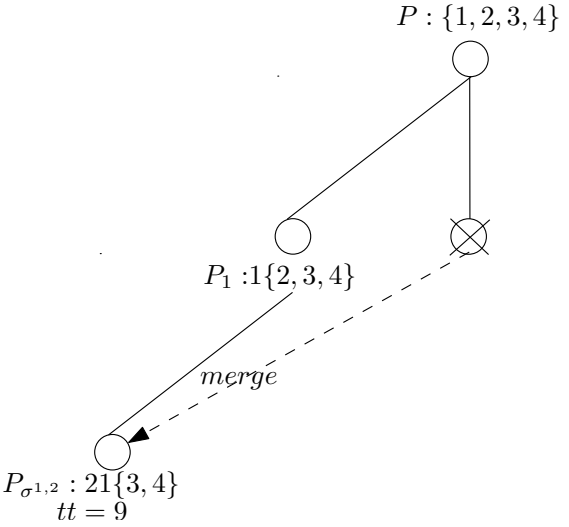
The algorithm continues its exploration in depth-first order, each time a leaf node is reached, the algorithm updates the current best solution. Another merging is done between $P_{4,2}$ and $P_{4,1,2}$, leading to prune $P_{4,2}$ (Figure 3.7d). Then, the algorithm goes on until all leaf nodes are explored and it returns the optimal solution given by $P_{4,1,2,3}$, corresponding to the sequence of $(2,3,4,1)$ (Figure 3.7e).

Notice that some nodes involved in merging have only one job to schedule, like $P_{4,2}$, but we still consider them as a non-solved subproblems in the exemple in order to illustrate the

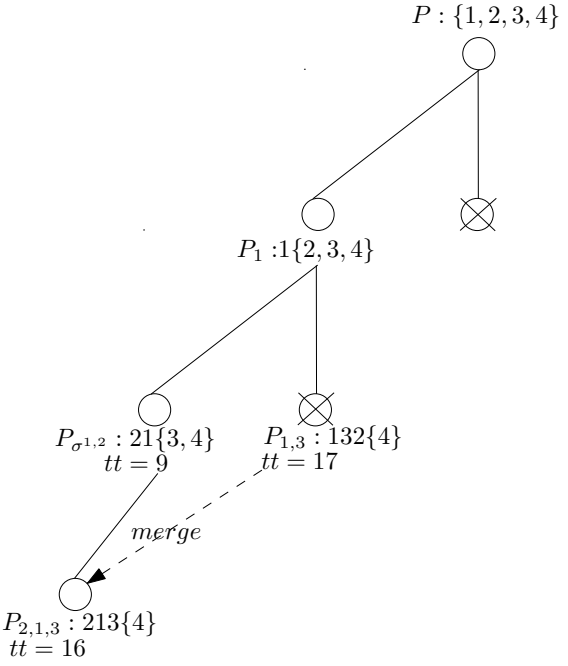
merging. Also notice that the `LEFT_MERGE` procedure should only be called on subproblems of enough size (greater than k). All the three nodes P, P_1 and P_4 have more than 2 jobs to schedule.


$P : \{1, 2, 3, 4\}$
$P_1 : 1\{2, 3, 4\}$
$P_2 : 21\{3, 4\}$
$tt = 9$
$P_{1,2} : 12\{3, 4\}$
$tt = 12$

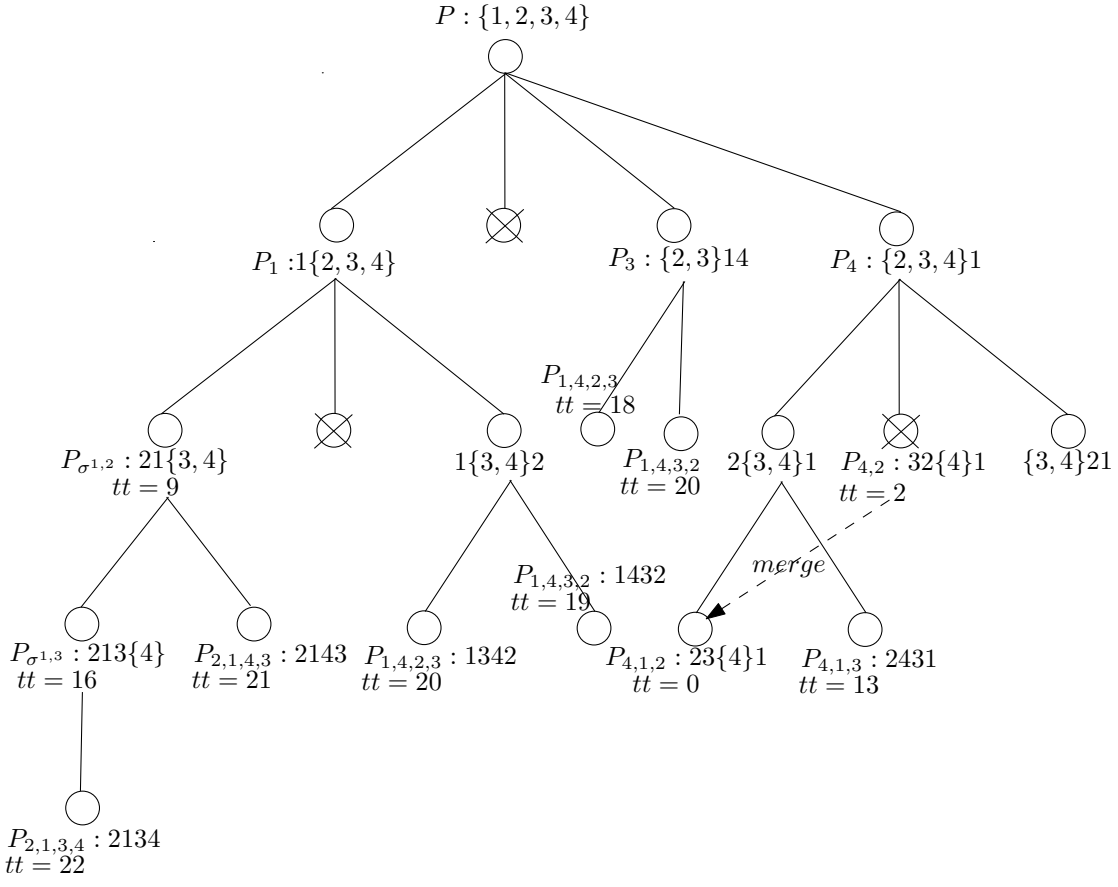
(a)


$P : \{1, 2, 3, 4\}$
$P_1 : 1\{2, 3, 4\}$
merge
$P_{\sigma^{1,2}} : 21\{3, 4\}$
$tt = 9$

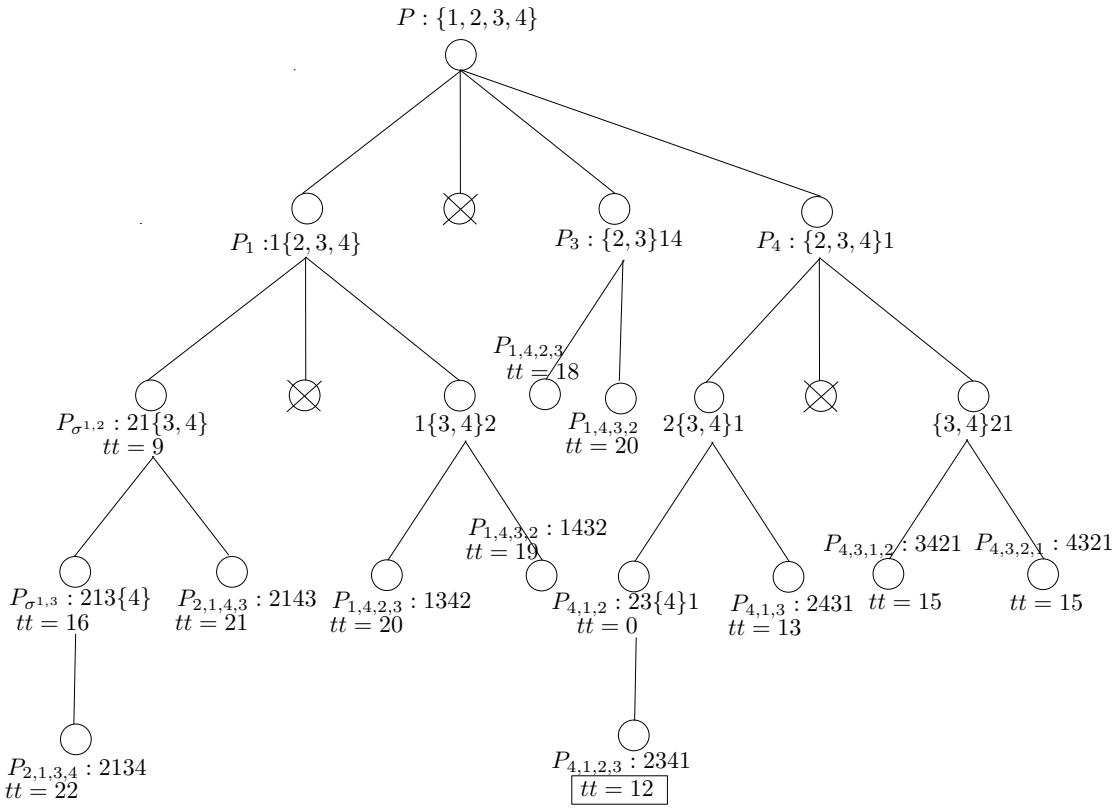
(b)


$P : \{1, 2, 3, 4\}$
$P_1 : 1\{2, 3, 4\}$
$P_{\sigma^{1,2}} : 21\{3, 4\}$
$tt = 9$
$P_{1,3} : 132\{4\}$
$tt = 17$
merge
$P_{2,1,3} : 213\{4\}$
$tt = 16$

(c)


$P : \{1, 2, 3, 4\}$
$P_1 : 1\{2, 3, 4\}$
$P_3 : \{2, 3\}14$
$P_4 : \{2, 3, 4\}1$
$P_{1,4,2,3}$
$tt = 18$
$P_{\sigma^{1,2}} : 21\{3, 4\}$
$tt = 9$
$1\{3, 4\}2$
$P_{1,4,3,2}$
$tt = 20$
$2\{3, 4\}1$
$P_{4,2} : 32\{4\}1$
$tt = 2$
$\{3, 4\}21$
merge
$P_{1,4,3,2} : 1432$
$tt = 19$
$P_{\sigma^{1,3}} : 213\{4\}$
$tt = 16$
$P_{2,1,4,3} : 2143$
$tt = 21$
$P_{1,4,2,3} : 1342$
$tt = 20$
$P_{4,1,2} : 23\{4\}1$
$tt = 0$
$P_{4,1,3} : 2431$
$tt = 13$
$P_{2,1,3,4} : 2134$
$tt = 22$

(d)



(e)

Figure 3.7: A sample instance solved by TTBR1 with left-merging integrated

3.3.2 Merging right-side branches

Due to the branching scheme, the merging of right-side branches involves a more complicated procedure than the merging of left-side branches. In the merging of left-side branches, it is possible to merge some nodes associated to problems P_ℓ with children nodes of P_1, while for the right-side branches, it is not possible to merge some nodes P_ℓ with children nodes of P_n. To see this, consider the situation depicted in Figure 3.8. If the merging can be applied in a symmetric way as on left-size branches, then the node P_{n-1} and the node $P_{n,n-1}$ should be concerned by a merging operation. However, due to Property 2, these two nodes do not share any common subproblem. Therefore, they can not be merged.

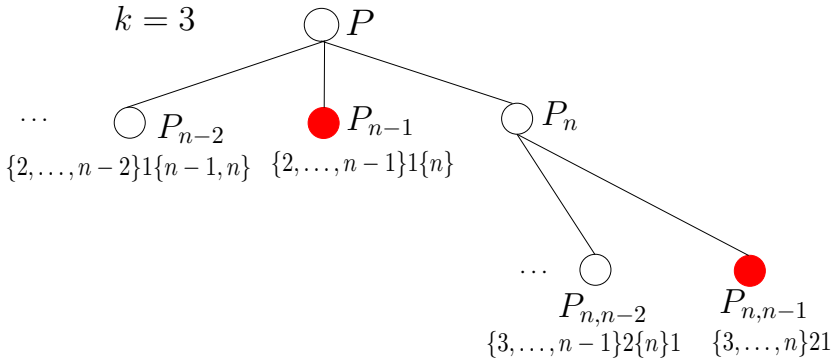


Figure 3.8: An failed attempt on merging right-side branches

However, another structure of merging exist: the children nodes of P_ℓ can be merged with the children nodes of P_n. Let us more formally introduce the right merging procedure and, again, let $k \leq \frac{n}{2}$ be the same constant parameter as used in the left merging.

Figure 3.9 shows an example on the structure of merging for the k right-side branches with $k = 3$. The root problem P consists in scheduling job set $\{1,\ldots,n\}$. Unlike left-side merging, the right-side merging is done horizontally for each level. Nodes that are involved in merging are colored. For instance, the black square nodes at level 1 can be merged together. This concerns three nodes: $P_{n-2,n-3}$, $P_{n-1,n-3}$, $P_{n,n-3}$. As indicated in the figure, all these three nodes contain an identical subproblem which consists in scheduling jobs $\{3,..,n-2\}$ from time 0. Also notice that apart from this subproblem, the number of remaining jobs to schedule is less than k. Therefore, these small number of jobs are fixed first, which then enables the merging of all the three nodes. The result of the merging should take the position of the right most node, which is $P_{n,n-3}$. Similarly, the black circle nodes at level 1 can be merged, the grey square nodes at level 2 can be merged and the grey circle nodes at level 2 can be merged. Notice that each right-side branch of P is expanded to a different depth which depends on the branching decision: for the purpose of right merging, a branch is no more expanded whenever the largest subproblem of the first child node is of size $(n-k-1)$. For example, in Figure 3.9, consider node P_{n-2} and its first child node $P_{n-2,1}$ which defines the subproblem $2\{3,..,n-2\}1\{n-1,n\}$. The largest subproblem in $P_{n-2,1}$ conerns jobs $\{3,..,n-2\}$ which has $(n-4) = (n-k-1)$ jobs. Then, the expansion of the branch rooted by $P_{n-2,1}$ is stopped. Notice that in the

implementation of the right merge, it is not mandatory to have the whole expansion of the search tree as indicated in Figure 3.9. But, this expansion eases the presentation of the mechanism and the analysis of the final complexity.

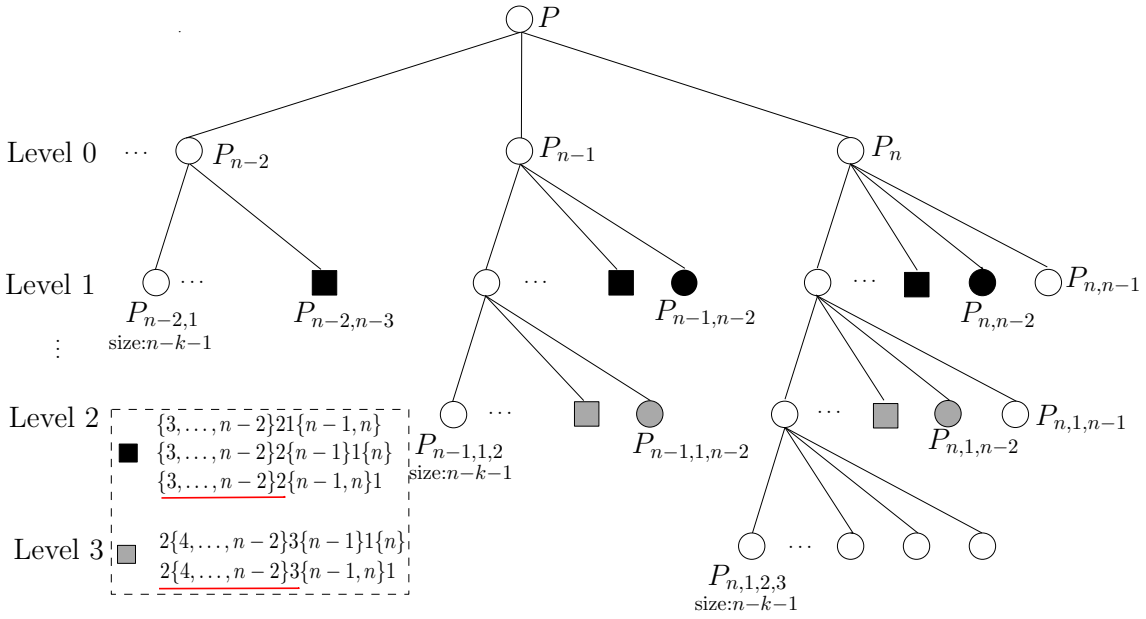


Figure 3.9: An example of right-side branches merging for $k = 3$

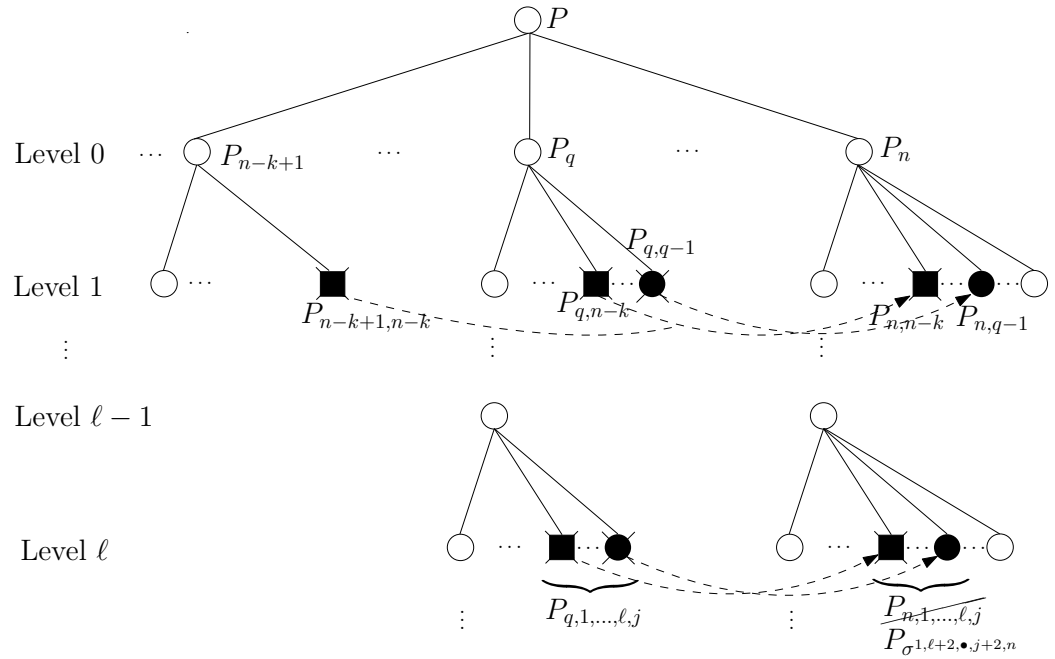
More generally, Figure 3.10 shows the right-side search tree and the content of the nodes involved in the merging for an arbitrary value of k.

The rest of this section intends to describe the merging by following the same lines as for left merging. We first extend the notation P_σ in the sense that σ may now contain placeholders. The i-th element of σ is either the position assigned to job i if i is fixed, or $\bullet$ if job i is not yet fixed. The $\bullet$ sign is used as placeholder, with its cardinality below indicating the number of consecutive $\bullet$. As an example, the problem $\{2, \ldots, n-1\}1n$ can now be denoted by $P_{n-1,\underset{n-2}{\bullet},n}$. The cardinality of $\bullet$ may be omitted whenever it is not important for the presentation or it can be easily deduced as in the above example. Note that this adapted notation eases the presentation of right merge while it has no impact on the validity of the results stated in the previous section.

Proposition 5. *Let P_σ be a problem to branch on. Let $j^*, \ell_b, \ell_e, \rho_1$ and ρ_2 be defined as in Proposition 3. Extending Corollary 1, problem P_σ has the following structure:*

$$\pi\{j^*, \ldots, j^* + \ell_e - \ell_b\}\gamma\Omega'$$

where π is defined as in Corollary 1 and γ is the sequence of jobs in positions $\rho_2, \ldots, \rho_3$ with $\rho_3 = \max\{i : i \geq \rho_2, \text{positions } \rho_2, \ldots, i \text{ are in } \sigma\}$ and Ω' the remaining subset of jobs to be scheduled after position ρ_3 (some of them can have been already scheduled). The merging procedure is applied on job set $\{j^*, \ldots, j^* + \ell_e - \ell_b\}$ preceded by a sequence of jobs π and followed by $\gamma\Omega'$.



$P_{q,1,...,\ell,j} : (2,\ldots,\ell{+}1)\{\ell{+}3,...,j{+}1\}(\ell{+}2)\{j{+}2,...,q\}1\{q{+}1,...,n\}$
$P_{\sigma^{1,\ell+2,\bullet,j+2,n}} : (2,\ldots,\ell{+}1)\{\ell{+}3,...,j{+}1\}\underset{\max\{j+1,n-k+\ell+1\}\leq q\leq n}{\texttt{BEST}}((\ell{+}2)\{j{+}2,...,q\}1\{q{+}1,...,n\})$

Figure 3.10: Generic right-side merging at the root node

Proof. The problem structure stated in Corollary 1 is refined on the part of Ω. Ω is split into two parts: γ and Ω' (see Figure 3.11). The motivation is that γ will be involved in the right merging, just like the role of π in left merging. □

Proposition 6 shed lights on how to merge the right side branches originated from the root node. We first introduce Definition 5 which will be used to identify a set of nodes to merge.

Definition 5. *Let $\mathcal{S}_{\ell,j}$ be a set of nodes, defined as*

$$\mathcal{S}_{\ell,j} = \left\{ P_\sigma : \begin{array}{l} |\sigma| = \ell + 2, \\ \max\{j+1, n-k+\ell+1\} \leq \\ \sigma_1 \leq n, \\ \sigma_i = i-1,\ \forall i \in \{2,\ldots,\ell+1\}, \\ \sigma_{\ell+2} = j \end{array} \right\}^1$$

with $0 \leq \ell \leq k-1$, $n-k \leq j \leq n-1$, and with σ_i referring to the position of job i in σ.

To understand the proposition intuitively, the reader can always refer to Figure 3.9 and Figure 3.10. Each set $\mathcal{S}_{\ell,j}$ represents a set of nodes with the same color and shape in Figure

[1] Placeholders do not count in the cardinality of σ

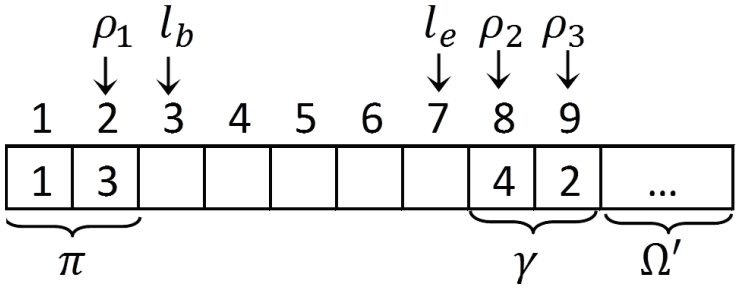


Figure 3.11: An example ($P_{1,9,2,8}$) for Proposition 5

3.9, that is, they can be merged together. The subscript ℓ in $\mathcal{S}_{\ell,j}$ indicates the level of nodes, more precisely the number of jobs branched on in π, and the subscript j determines the "shape" of nodes at that level, more precisely the position of the first job in γ. As an example, all black squares in Figure 3.9 are in $\mathcal{S}_{0,n-3} = \{P_{n,n-3}, P_{n-1,n-3}, P_{n-2,n-3}\}$.

We now introduce Proposition 6.

Proposition 6. *For each problem in the set $\mathcal{S}_{\ell,j}$, we have the two following properties:*

1. *The solution of problems in $\mathcal{S}_{\ell,j}$ involves the solution of a common subproblem which consists in scheduling job set $\{\ell{+}3, ..., j{+}1\}$ starting at time $t_\ell{=}\sum_{i=2}^{\ell+1} p_i$.*
2. *For any problem in $\mathcal{S}_{\ell,j}$, at most $k{+}1$ jobs have to be scheduled after job set $\{\ell{+}3, ..., j{+}1\}$.*

Proof. As each problem $P_\sigma{\in}\mathcal{S}_{\ell,j}$ is defined by $(2, \ldots, \ell{+}1)\{\ell{+}3, ..., j{+}1\}(\ell{+}2)\{j{+}2, ..., \sigma_1\}1\{\sigma_1{+}1, ..., n\}$, the first part of the proposition is straightforward.
Besides, the second part can be simply established by counting the number of jobs to be scheduled after job set $\{\ell{+}3, ..., j{+}1\}$ when j is minimal, *i.e.* when $j{=}n{-}k$. In this case, $(\ell{+}2)\{j{+}2, ..., \sigma_1\}1\{\sigma_1{+}1, ..., n\}$ contains $k{+}1$ jobs. □

The above proposition highlights the fact that some nodes can be merged as soon as they share the same initial subproblem to be solved. More precisely, at most $(k - \ell - 1)$ nodes associated to problems $P_{q,1..\ell,j}$, $\max\{j{+}1, n{-}k{+}\ell{+}1\} \leq q \leq (n{-}1)$, can be merged with the node associated to problem $P_{n,1..\ell,j}$, $\forall j = (n-k), ..., (n-1)$. The node $P_{n,1..\ell,j}$ is replaced in the search tree by the node $P_{\sigma^{1,\ell+2,\bullet,j+2,n}}$ defined as follows (Figure 3.10):

- $\{\ell + 3, ..., j + 1\}$ is the set of jobs on which it remains to branch.
- Let $\sigma^{1,\ell+2,\bullet,j+2,n}$ be the sequence containing positions of jobs $\{1, \ldots, \ell{+}2, j{+}2, \ldots, n\}$ and placeholders for the other jobs, that leads to the best jobs permutation among $(\ell + 2)\{j + 2, ..., q\}1\{q + 1, ..., n\}$, $\max\{j + 1, n - k + \ell + 1\} \leq q \leq n$. This involves the solution of at most k problems of size at most $k + 1$ (in $\mathcal{O}^*(k \times 2.4143^{k+1})$ time by TTBR2) and the determination of the best of the computed sequences knowing

that all of them start at time t, namely the sum of the jobs processing times in $(2, \ldots, \ell+1)\{\ell+3, ..., j+1\}$.

The merging process described above is applied at the root node, while an analogous merging can be applied at any node of the tree. With respect to the root node, the only additional consideration is that the right-side branches of a general node may have already been modified by previous merging operations. As an example, let us consider Figure 3.12. It shows that, due to the merging operations performed from P, the right-side branches of P_n may not be the subproblems induced by the branching scheme. However, it can be shown in a similar way as for left-merge, that the merging can still be applied.

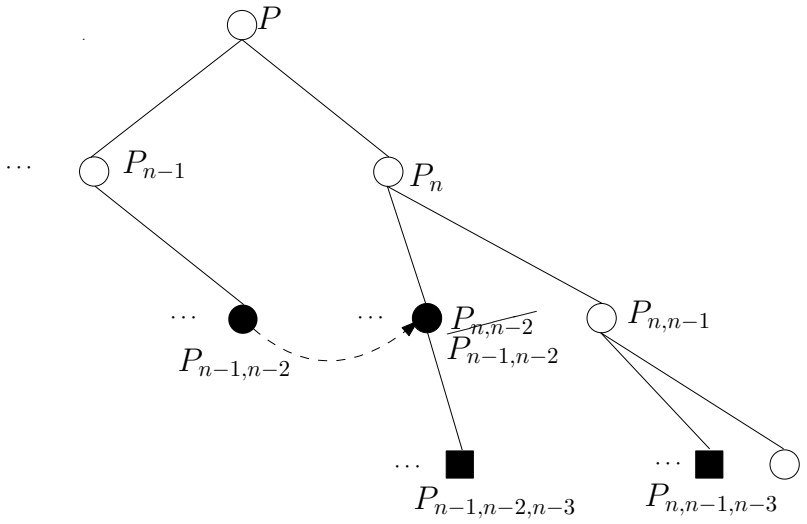


Figure 3.12: The right branches of P_n were modified by the right-merging from P

In order to define the branching scheme used with the `RIGHT_MERGE` procedure, a data structure $\mathcal{R}_\sigma$ is associated to a problem P_σ. It represents a list of subproblems that result from a previous merging and are now the k right-side children nodes of P_σ. When a merging operation sets the k right-side children nodes of P_σ to $P_{\sigma^{n-k+1}}, ..., P_{\sigma^n}$, we set $\mathcal{R}_\sigma = \{P_{\sigma^{n-k+1}}, ..., P_{\sigma^n}\}$, otherwise we have $\mathcal{R}_\sigma = \emptyset$. As a conclusion, the following branching scheme for an arbitrary node of the tree is defined. It is an extension of the branching scheme introduced in Definition 4 for the left merging.

Definition 6. *The branching scheme for an arbitrary node P_σ is defined as follows:*

- *If $\mathcal{R}_\sigma = \emptyset$, use the branching scheme defined in Definition 4;*
- *If $\mathcal{L}_\sigma = \emptyset$ and $\mathcal{R}_\sigma \neq \emptyset$, branch on the longest job in the available positions from the $1st$ to the $(n-k)$-th, then extract problems from $\mathcal{R}_\sigma$ as the last k branches.*
- *If $\mathcal{L}_\sigma \neq \emptyset$ and $\mathcal{R}_\sigma \neq \emptyset$, extract problems from $\mathcal{L}_\sigma$ as the first $(k-1)$ branches, then branch on the longest job in the available positions from the k-th to the $n-k$-th, finally extract problems from $\mathcal{R}_\sigma$ as the last k branches.*

This branching scheme generalizes and replaces, the one introduced in Definition 4 and it will be referred to as ***improved branching*** *from now on.*

Proposition 7 states the validity of merging a general node, which extends the result in Proposition 6.

Proposition 7. *Let P_σ be an arbitrary problem and let $\pi, j^*, \ell_b, \ell_e, \gamma, \Omega'$ be computed relatively to P_σ according to Proposition 5. If $\mathcal{R}_\sigma=\emptyset$, the right merging on P_σ can be easily performed by considering P_σ as a new root problem. Suppose $\mathcal{R}_\sigma\neq\emptyset$, the q-th child node P_{σ^q} is extracted from $\mathcal{R}_\sigma$, $\forall n'-k+1\leq q\leq n'$, where $n'=\ell_e-\ell_b+1$ is the number of children nodes of P_σ. The structure of P_{σ^q} is $\pi\{j^*+1,...,j^*+q-1\}\gamma^q\Omega'$.*

For $0\leq\ell\leq k-1$ and $n'-k\leq j\leq n'-1$, the following conditions hold:

1. *Problems in $\mathcal{S}^\sigma_{\ell,j}$ have the following structure: $\pi(j^*+1,\ldots,j^*+\ell)\{j^*+\ell+2,...,j^*+j\}(j^*+\ell+1)\{j^*+j+1,...,j^*+q-1\}\gamma^q\Omega'$ with q varies from $\max\{j+1, n-k+\ell+1\}$ to n'.*
2. *The solution of all problems in $\mathcal{S}^\sigma_{\ell,j}$ involves the scheduling of a job set $\{j^*+j+1,...,j^*+q-1\}$, $\max\{j+1,n-k+\ell+1\}\leq q\leq n'$, which is of size less than k. Besides, for all problems in $\mathcal{S}^\sigma_{\ell,j}$ it is required to solve a common subproblem made of job set $\{j^*+\ell+2,...,j^*+j\}$ starting after $\pi(j^*+1,\ldots,j^*+\ell)$ and before $(j^*+\ell+1)\{j^*+j+1,...,j^*+q-1\}\gamma^q\Omega'$.*

Proof. The proof is similar to the one of Proposition 4. The first part of the statement follows directly from Definition 6 and simply defines the structure of the children nodes of P_σ. For the second part, it is necessary to prove that $\{j^*+j+1,...,j^*+q-1\}\gamma^q$ consists of the same jobs for any valid value of q. Actually, since right-merging only merges nodes that have common jobs fixed after the unscheduled jobs, the jobs present in $\{j^*+j+1,...,j^*+q-1\}\gamma^q$ and the jobs present in $\{j^*+j+1,...,j^*+q-1\}j^*\{j^*+q,...,j^*+n'-1\}\gamma$, $\max\{j+1,n-k+\ell+1\}\leq q\leq n'$, must be the same, which proves the statement. □

Analogously to the scenario at the root node, given the values of ℓ and j, all the problems in $\mathcal{S}^\sigma_{\ell,j}$ can be merged. More precisely, we rewrite σ as $\alpha\underset{n'}{\bullet}\beta$ where α is the sequence of positions assigned to jobs $\{1,\ldots,j^*-1\}$, $\underset{n'}{\bullet}$ refers to the job set to branch on and β contains the positions assigned to the rest of jobs. At most $(k-\ell-1)$ nodes associated to problems $P_{\alpha,\ell_b+q-1,\ell_b..\ell_b+\ell-1,\ell_b+j-1,\bullet,\beta}$, with $\max\{j+1,n'-k+\ell+1\}\leq q\leq n'-1$, can be merged with the node associated to problem $P_{\alpha,\ell_e,\ell_b..\ell_b+\ell-1,\ell_b+j-1,\bullet,\beta}$.

Node $P_{\alpha,\ell_e,\ell_b..\ell_b+\ell-1,\ell_b+j-1,\bullet,\beta}$ is replaced in the search tree by node $P_{\alpha,\sigma^{\ell,\ell_b,j},\bullet,\beta}$ defined as follows:

- $\{j^*+\ell+2,...,j^*+j\}$ is the set of jobs on which it remains to branch.
- Let $\sigma^{\ell,\ell_b,j}$ be the sequence of positions among

$$\{(\ell_b+q-1,\ell_b..\ell_b+\ell-1,\ell_b+j-1) : \max\{j+1,n'-k+\ell+1\}\leq q\leq n'-1\}$$

  associated to the best permutation of jobs in $(j^*+\ell+1)\{j^*+j+1,...,j^*+q-1\}\gamma^q$, $\forall\max\{j+1,n'-k+\ell+1\}\leq q\leq n'$. This involves the solution of k problems of size at most $k+1$ (in $\mathcal{O}^*(k\times 2.4143^{k+1})$ time by TTBR2) and the determination of the best

of the computed sequences knowing that all of them start at time t, i.e. the sum of the jobs processing times in $\pi(j^*+1,\ldots,j^*+\ell)\{j^*+\ell+2,...,j^*+j\}$.

The `RIGHT_MERGE` procedure is presented in Algorithm 3. Notice that, similarly to the `LEFT_MERGE` procedure, this algorithm takes as input one problem P_σ and provides as an output a set of nodes to branch on, which replaces all its k right-side children nodes of P_σ. It is important to notice that the `LEFT_MERGE` procedure is also integrated.

A procedure `MERGE_RIGHT_NODES` (Algorithm 4) is invoked to perform the right merging for each level $\ell = 0,...,k-1$ in a recursive way. The initial inputs of this procedure (line 13 in `RIGHT_MERGE`) are the problem P_σ and the list of its k right-side children nodes, denoted by *rnodes*. They are created according to the improved branching (lines 4-12 of Algorithm 3). Besides, the output is a list Q containing the problems to branch on after merging. In the first call to `MERGE_RIGHT_NODES`, the left merge is applied to the first element of *rnodes* (line 2), all the children nodes of nodes in *rnodes* not involved in right nor left merging, are added to Q (lines 3-7). This is also the case for the result of the right merging operations at the current level (lines 8-11). In Algorithm 4, the value of r indicates the current size of *rnodes*. It is reduced by one at each recursive call and the value $(k-r)$ identifies the current level with respect to P_σ. As a consequence, each right merging operation consists in finding the problem with the best total tardiness value on its fixed part, among the ones in set $\mathcal{S}^\sigma_{k-r,j}$. This is performed by the `BEST` function (line 10 of `MERGE_RIGHT_NODES`) which extends the one called in Algorithm 2 by taking at most k subproblems as input and returning the dominating one.

The `MERGE_RIGHT_NODES` procedure is then called recursively on the list containing the first child node of the 2nd to r-th node in *rnodes* (lines 13-17). Note that the procedure `LEFT_MERGE` is applied to every node in *rnodes* except the last one. In fact, for any specific level, the last node in *rnodes* belongs to the last branch of P_σ, which is $P_{\sigma,l_b+n-1,\bullet,\beta}$. Since $P_{\sigma,l_b+n-1,\bullet,\beta}$ is put into Q at line 14 of `RIGHT_MERGE`, it means that this node will be re-processed later and `LEFT_MERGE` will be called on it at that moment. Since the recursive call of `MERGE_RIGHT_NODES` (line 18) will merge some nodes to the right-side children nodes of $P_{\alpha,\ell_b,\underset{n_r-1}{\bullet},\beta^r}$, the latter one must be added to the list $\mathcal{L}$ of $P_{\alpha,\underset{n_r}{\bullet},\beta^r}$ (line 19). In addition, since we defined $\mathcal{L}$ as a list of size either 0 or $(k-1)$, lines 20-24 add the other $(k-2)$ nodes to $\mathcal{L}_{\alpha,\underset{n_r}{\bullet},\beta^r}$.

It is also important to notice the fact that a node may have its $\mathcal{L}$ or $\mathcal{R}$ structures non-empty, if and only if it is the first or last child node of its parent node. A direct result is that only one node among those involved in a merging may have its $\mathcal{L}$ or $\mathcal{R}$ non-empty. In this case, these structures need to be associated to the resulting node. The reader can always refer to Figure 3.9 for a more intuitive representation.

Lemma 13. *The `RIGHT_MERGE` procedure returns a list of $\mathcal{O}(n)$ nodes in polynomial time and space.*
The solution of the problems associated to these nodes involves the solution of 1 *subproblem of size* $(n-1)$*, of* $(k-1)$ *subproblems of size* $(n-k-1)$*, and subproblems of size* i *and* $(n_q-(k-r)-i-1)$*,* $\forall r=2,...,k; q=1,...,(r-1); i=k,...,(n-2k+r-2)$*, where* n_q *is defined in Algorithm 4 (`MERGE_RIGHT_NODES`).*

Algorithm 3 `RIGHT_MERGE` Procedure

Input: $P_\sigma = P_{\alpha,\underset{n}{\bullet},\beta}$ a problem of size n, with ℓ_b, j^* computed according to Proposition 3
Output: Q : a list of problems to branch on after merging

```
1: function RIGHT_MERGE(P_σ)
2:     Q ← ∅
3:     nodes ← ∅
4:     if R_σ = ∅ then
5:         for q = n−k+1 to n do
6:             Create P_{α,ℓ_b+q−1,•,β} by branching
7:             δ ← the sequence of positions of jobs {j*+q, ..., j*+n−1} fixed by TTBR2
8:             nodes ← nodes∪P_{α,ℓ_b+q−1,•,δ,β}
9:         end for
10:    else
11:        nodes ← R_σ
12:    end if
13:    Q ← Q∪MERGE_RIGHT_NODES(nodes, P_σ)
14:    Q ← Q∪nodes[k]                          ▷ The last node will be re-processed
15:    return Q
16: end function
```

Proof. The first part of the result follows directly from Algorithm 3. The only lines where nodes are added to Q in `RIGHT_MERGE` are lines 13-14. In line 14, only one problem is added to Q, thus it needs to be proved that the call on `MERGE_RIGHT_NODES` (line 13) returns $\mathcal{O}(n)$ nodes. This can be computed by analysing the lines 2-7 of Algorithm 4. Considering all recursive calls, the total number of nodes returned by `MERGE_RIGHT_NODES` is $(\sum_{i=1}^{k-1}(k-i)(n-2k-i))+k-1$ which yields $\mathcal{O}(n)$. The number of all nodes considered in right merging is bounded by a linear function on n. Furthermore, all the operations associated to the nodes (merging, creation, etc) have a polynomial cost. As a consequence, Algorithm 3 runs in polynomial time and space.

Regarding the sizes of the subproblems returned by `RIGHT_MERGE`, the node added in line 14 of Algorithm 3 contains one subproblem of size $(n-1)$, corresponding to branching the longest job on the last available position. Note that even though the subtree of this node has been modified by merging operations, its solution time is still bounded by the time for solving a subproblem of size $(n-1)$.

Then, the problems added by the call to `MERGE_RIGHT_NODES` are added to Q. In line 2 of Algorithm 4, the size of the problem returned by `LEFT_MERGE` is reduced by one unit when compared to the input problem which is of size $(n-k-(k-r))$. Note that $(k-r)$ is the current level with respect to the node tackled by Algorithm 4. As a consequence, the size of the resulting subproblem is $(n-k-(k-r)-1)$. Note that this line is executed $(k-1)$ times, $\forall r = k, \ldots, 2$, corresponding to the number of calls to `MERGE_RIGHT_NODES`. In line 5 of Algorithm 4, the list of nodes which are not involved in any merging operation are added to Q. This corresponds to pairs of problems of size i and $(n_q-(k-r)-i-1)$, $\forall i = k, ..., (n-k-1)$ and this proves the last part of the lemma. □

Algorithm 4 `MERGE_RIGHT_NODES` Procedure

Input: $rnodes = [P_{\alpha,\bullet_{n_1},\beta^1}, \ldots, P_{\alpha,\bullet_{n_r},\beta^r}]$, ordered list of r last children nodes with ℓ_b defined on any node in $rnodes$. $|\alpha|+1$ is the job to branch on and $n_r = n_1 + r - 1$.

Output: Q, a list of problems to branch on after merging

1: **function** `MERGE_RIGHT_NODES`$(rnodes, P_\sigma)$
2: $Q \leftarrow$ `LEFT_MERGE`$(P_{\alpha,\bullet_{n_1},\beta^1})$
3: **for** $q = 1$ to $r-1$ **do**
4: **for** $j = \ell_b + \min(k, \lfloor n/2 \rfloor)$ to $\ell_b + n_1 - 1$ **do**
5: $Q \leftarrow Q \cup P_{\alpha,j,\bullet_{n_q-1},\beta^q}$
6: **end for**
7: **end for**
8: **for** $j = \ell_b + n_1$ to $\ell_b + n_r$ **do**
9: Solve all the subproblems of size less than k in $\mathcal{S}^\sigma_{k-r,j}$
10: $\mathcal{R}_{\alpha,\bullet_{n_r},\beta^r} \leftarrow \mathcal{R}_{\alpha,\bullet_{n_r},\beta^r} \cup$ `BEST`$(\mathcal{S}^\sigma_{k-r,j})$
11: **end for**
12: **if** $r > 2$ **then**
13: $newnodes \leftarrow \emptyset$
14: **for** $q = 2$ to $r-1$ **do**
15: $newnodes \leftarrow newnodes \cup$ `LEFT_MERGE`$(P_{\alpha,\bullet_{n_q},\beta^q})$
16: **end for**
17: $newnodes \leftarrow newnodes \cup P_{\alpha,\ell_b,\bullet_{n_r-1},\beta^r}$
18: $Q \leftarrow Q \cup$ `MERGE_RIGHT_NODES`$(newnodes, P_\sigma)$
19: $\mathcal{L}_{\alpha,\bullet_{n_r},\beta^r} \leftarrow P_{\alpha,\ell_b,\bullet_{n_r-1},\beta^r}$
20: **for** $q = 2$ to $k-1$ **do**
21: Create $P_{\alpha,\ell_b+q-1,\bullet_{n_r-1},\beta^r}$ by branching
22: $\delta \leftarrow$ the sequence of positions of jobs $\{|\alpha|+2, \ldots, |\alpha|+q\}$ fixed by TTBR2
23: $\mathcal{L}_{\alpha,\bullet_{n_r},\beta^r} \leftarrow \mathcal{L}_{\alpha,\bullet_{n_r},\beta^r} \cup P_{\alpha,\ell_b+q-1,\delta,\bullet_{n_r-1},\beta^r}$
24: **end for**
25: **end if**
26: **return** Q
27: **end function**

Lemma 14 considers non-worst-case scenarios.

Lemma 14. *When right merging is incorporated into the branching scheme of TTBR, instances such that $LPT = EDD$ correspond to worst-case instances.*

Proof. The proof follows a similar reasoning to the proof of Lemma 12. We analyze the time reduction obtained from the last k subtrees of the current node. The following cases are considered:

1. $1 = [j]$, $j \leq (n - k + 1)$ and $2 = [j']$, $j' \leq (n - k)$;
2. $1 = [j]$, $j \leq (n - k + 1)$ and $2 = [j']$, $j' > (n - k)$;
3. $1 = [j]$, $j > (n - k + 1)$.

We refer the reader to Figure 3.10 for a better understanding of the proof. Since the structure of right merging is the same at different levels (except level 0) of the tree, it is sufficient to consider the time reduction from level 0 and level 1. We denote the resulting time reduction by $TR1$, $TR2$ and $TR3$ for each of the three cases, respectively. We also note $TR_{LPT=EDD}$ the time reduction corresponding to the case $LPT = EDD$.

In case 1, all black nodes at level 1 of Figure 3.10 are created and merged to one. Therefore, the corresponding time reduction $TR1 \geq TR_{LPT=EDD} = \sum_{i=1}^{k-1} iT(n - i - 2)$.

In case 2, some black nodes at level 1 of Figure 3.10 are not created due to Property 2, while the black nodes that remain can still be merged to the last subtree. Therefore, $TR2 \geq TR1$.

In case 3, the subtrees rooted by $\{P_1, ..., P_{j-1}\}$ are not created due to Property 2. This is obviously a stronger reduction than only merging some nodes from inside these subtrees. In addition, for subtrees that remain except the last one, i.e., those rooted by $\{P_j, ..., P_n\}$, time reduction is still guaranteed by, either right merging or the non-creation of some nodes due to Property 2, depending on the position of job 2 in EDD ordering.

In other words, if $LPT \neq EDD$ then the number of nodes in $\mathcal{S}^{\sigma}_{\ell,j}$ (defined in Proposition 7) can be less, since some nodes may not be created due to Property 2. However, all the nodes inside $\mathcal{S}^{\sigma}_{\ell,j}$ can still be merged to one except when $\mathcal{S}^{\sigma}_{\ell,j}$ is empty. In either case, we can achieve at least the same reduction as the case of $LPT = EDD$. This reasoning obviously holds when extending the consideration to all levels of the tree and to all recursions. Therefore, $LPT = EDD$ is the worst-case scenario. □

3.3.2.1 A working example for Right Merge

In order to better illustrate the merging operation on right-side branches, an example is provided in this section. The input data ($n = 6$) is given in Table 3.2 and the value of k is chosen as 3.

The solution of the instance is depicted in Figure 3.13. The indicated partial total tardiness value ($tt = \sum T_j$) is computed on the jobs that are fixed before and after the unscheduled jobs. The applied algorithm is TTBR1 with `RIGHT_MERGE` integrated. At the beginning, the root node P is the current node, and `RIGHT_MERGE` is called on it. Line

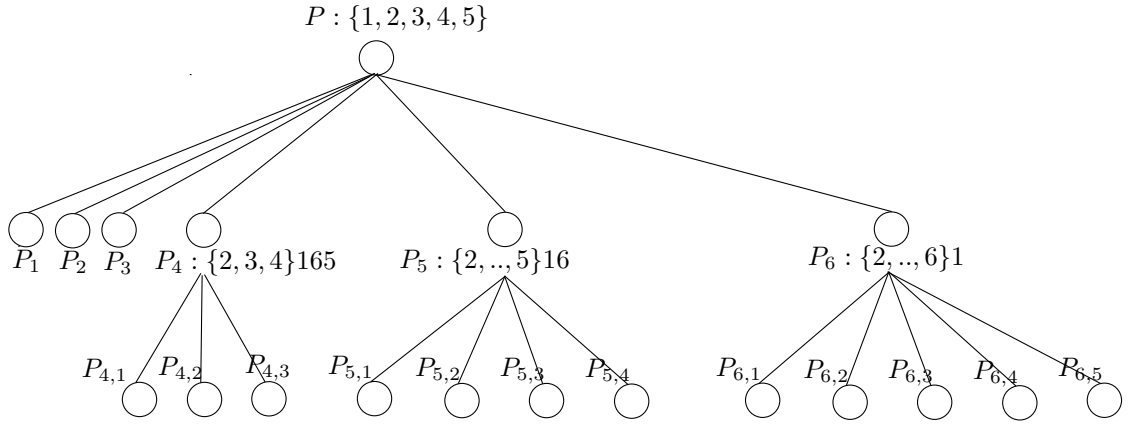
i	1	2	3	4	5	6
p_i	11	10	7	4	2	1
d_i	10	12	14	16	18	19

Table 3.2: A sample instance

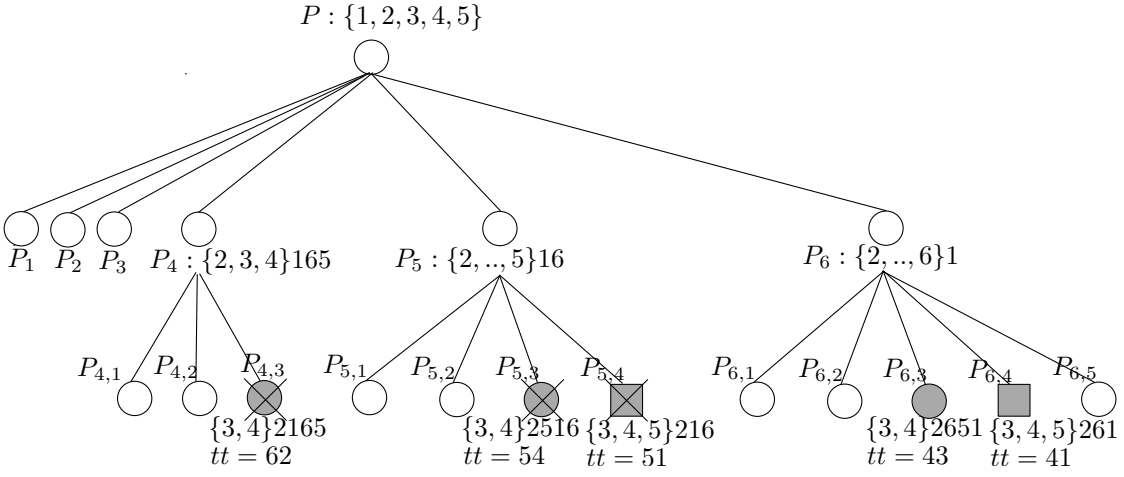
1-12 in Algorithm 3 creates child nodes in order to prepare the merging, as shown in 3.13a. Notice that the small subproblem involving jobs $\{5, 6\}$ inside P_4 is solved directly to obtain the optimal sequence which is $(6, 5)$. Line 13 calls the `MERGE_RIGHT_NODES` procedure with arguments (P_4, P_5, P_6).

In `MERGE_RIGHT_NODES` procedure (Algorithm 4), line 2 means to apply `LEFT_MERGE` on P_4. This is not performed in our exemple since P_4 only have 3 jobs to schedule and k is supposed to verify $2 \leq k \leq n/2$. Therefore, we add directly $P_{4,1}$ to $\mathcal{Q}$. Line 3-7 means to add all "middle" nodes that are not involved by merging hence should be opened later as independent subproblems. This includes $P_{4,2}$ only. Notice that $\ell_b = 1$ at line 4. Lines 8-11 do the merging. As shown in Figure 3.7b, nodes $P_{4,3}$, $P_{5,3}$ are merged with $P_{6,3}$ and the node $P_{5,4}$ is merged with $P_{6,4}$. The dominant nodes are $P_{6,3}$ and $P_{6,3}$, hence nodes nodes $P_{4,3}$, $P_{5,3}$ and $P_{5,4}$ are cut. After this, at line 15, `LEFT_MERGE` is applied on P_5 (Figure 3.13c) then the `MERGE_RIGHT_NODES` procedure is called in a recursive manner on nodes $P_{5,1}$ and $P_{6,1}$. The nodes $P_{5,1,3}$ and $P_{5,1,4}$ are finally cut due to merging (Figure 3.13d).

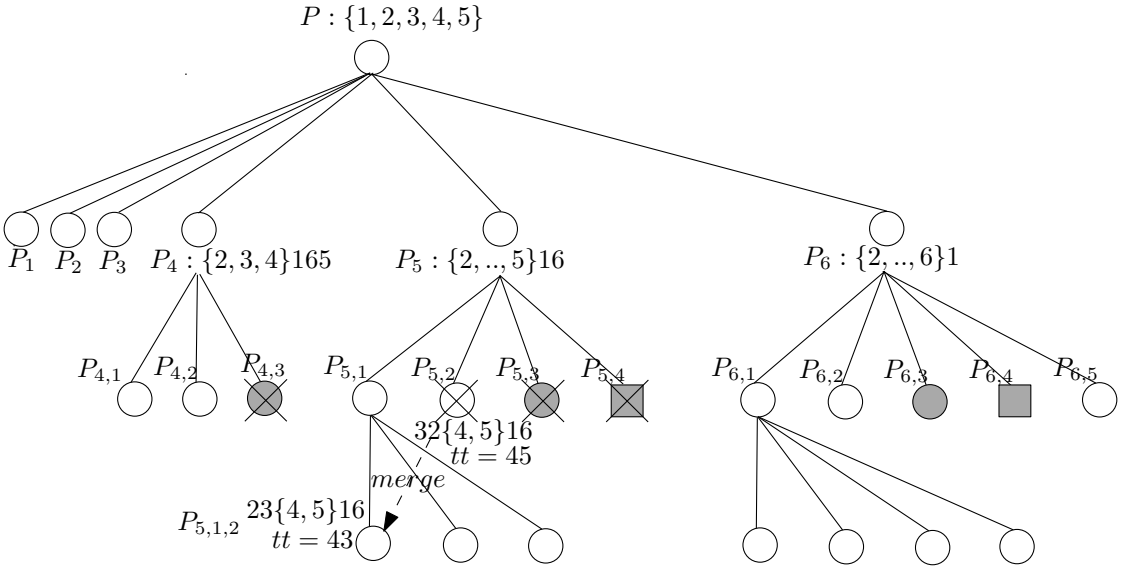
Then the algorithm continues in a recursive way. In depth-first order, the node P_1 is the next to open.



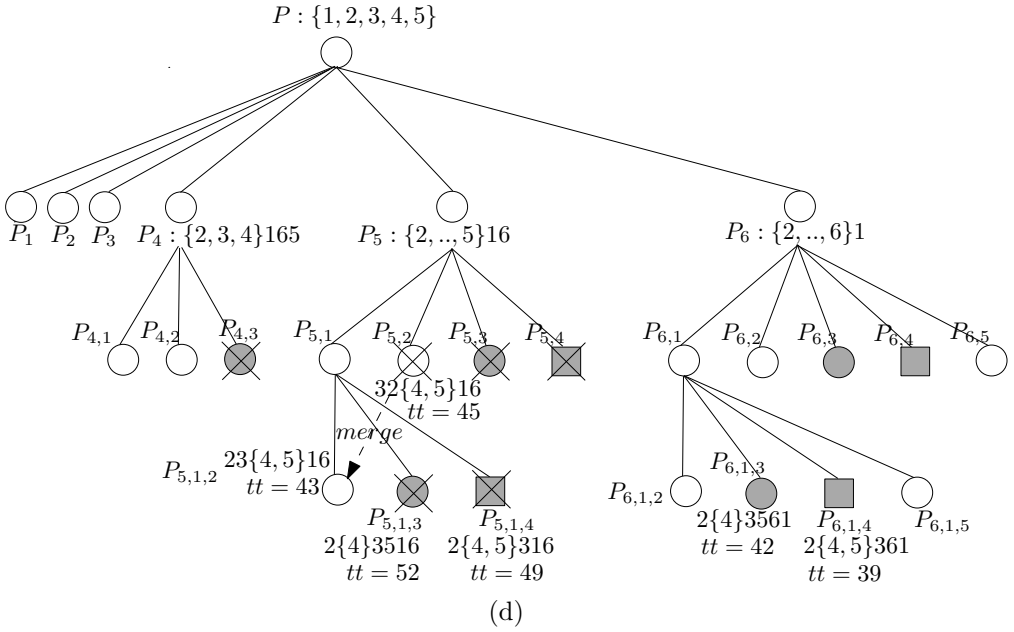
(a)


$P : \{1, 2, 3, 4, 5\}$
P_1 P_2 P_3 $P_4 : \{2, 3, 4\}165$ $P_5 : \{2, .., 5\}16$ $P_6 : \{2, .., 6\}1$
$P_{4,1}$ $P_{4,2}$ $P_{4,3}$ $P_{5,1}$ $P_{5,2}$ $P_{5,3}$ $P_{5,4}$ $P_{6,1}$ $P_{6,2}$ $P_{6,3}$ $P_{6,4}$ $P_{6,5}$
$\{3, 4\}2165$ $tt = 62$
$\{3, 4\}2516$ $tt = 54$
$\{3, 4, 5\}216$ $tt = 51$
$\{3, 4\}2651$ $tt = 43$
$\{3, 4, 5\}261$ $tt = 41$

(b)


$P : \{1, 2, 3, 4, 5\}$
P_1 P_2 P_3 $P_4 : \{2, 3, 4\}165$ $P_5 : \{2, .., 5\}16$ $P_6 : \{2, .., 6\}1$
$P_{4,1}$ $P_{4,2}$ $P_{4,3}$ $P_{5,1}$ $P_{5,2}$ $P_{5,3}$ $P_{5,4}$ $P_{6,1}$ $P_{6,2}$ $P_{6,3}$ $P_{6,4}$ $P_{6,5}$
$32\{4, 5\}16$
$tt = 45$
$merge$
$P_{5,1,2}$ $23\{4, 5\}16$
$tt = 43$

(c)



(d)

Figure 3.13: A sample instance solved by TTBR1 with right-merging integrated

3.3.3 Complete algorithm and analysis

We are now ready to define the main procedure `TTBM` (Total Tardiness *Branch & Merge*), stated in Algorithm 5 which is called on the initial input problem $P : \{1, ..., n\}$. The algorithm has a similar recursive structure as TTBR1. However, each time a node is opened, the sub-branches required for the merging operations are generated, the subproblems of size less than k are solved and the procedures `LEFT_MERGE` and `RIGHT_MERGE` are called. Then, the algorithm proceeds recursively by extracting the next node from Q with a depth-first strategy and terminates when Q is empty.

Algorithm 5 Total Tardiness Branch and Merge (TTBM)

Input: $P : \{1, ..., n\}$: input problem of size n
$\frac{n}{2} \geq k \geq 2$: an integer constant
Output: *seqOpt*: an optimal sequence of jobs

function TTBM(P,k)
  $Q \leftarrow P$
  seqOpt $\leftarrow$ the EDD sequence of jobs
  while $Q \neq \emptyset$ **do**
    $P^* \leftarrow$ extract next problem from Q (depth-first order)
    if the size of $P^* < 2k$ **then**
      Solve P^* by calling TTBR2
    end if
    if all jobs $\{1, ..., n\}$ are fixed in P^* **then**
      seqCurrent $\leftarrow$ the solution defined by P^*
      seqOpt $\leftarrow$ best solution between *seqOpt* and *seqCurrent*
    else
      $Q \leftarrow Q \cup$ `LEFT_MERGE`(P^*) ▷ Left-side nodes
      for $i = k + 1, ..., n - k$ **do**
        Create the i-th child node P_i by branching scheme of TTBR1
        $Q \leftarrow Q \cup P_i$
      end for
      $Q \leftarrow$ `RIGHT_MERGE`(P^*) ▷ Right-side nodes
    end if
  end while
  return *seqOpt*
end function

The complexity of the algorithm depends on the value k. The higher it is, the more subproblems can be merged and the better is the worst-case time complexity of the algorithm. Figure 3.14 demonstrates the reduction that can be obtained by TTBM on each recursive call. It can help complexity analysis of the algorithm. Proposition 8 provides details about the time complexity of the proposed algorithm.

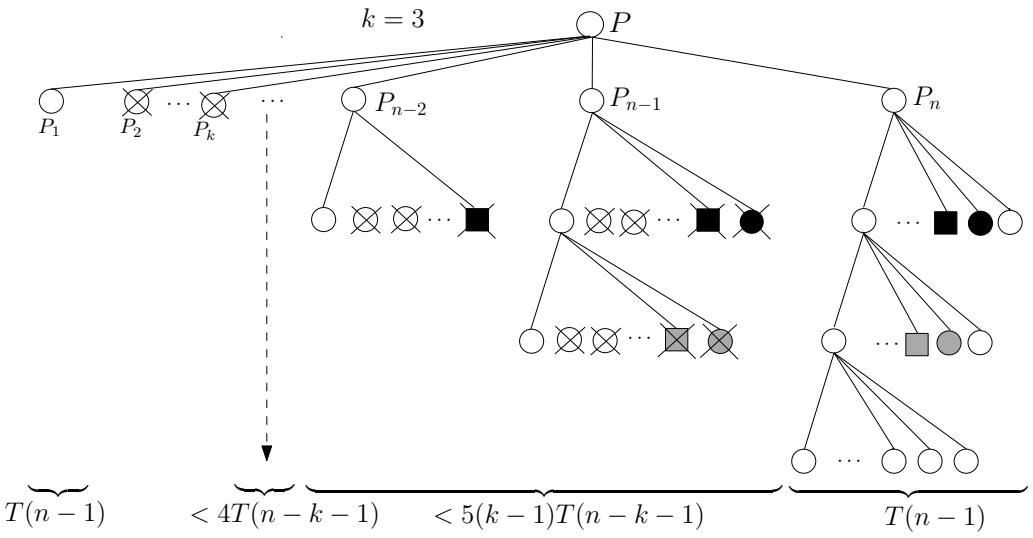


Figure 3.14: Reduction obtained by merging

Proposition 8. *Algorithm TTBM runs in* $\mathcal{O}^*((2+\epsilon)^n)$ *time and polynomial space, where* $\epsilon \to 0$ *when* $k \to \infty$.

Proof. The proof is based on the analysis of the number and the size of the subproblems put in Q when a single problem P^* is expanded. As a consequence of Lemma 12 and Lemma 14, TTBM induces the following recursion:

$$\begin{aligned} T(n) =& 2T(n-1) + 2T(n-k-1) + ... + 2T(k) \\ &+ \sum_{r=2}^{k} \sum_{q=1}^{r-1} \sum_{i=k}^{n_1-(k-r)-2} (T(i) + T(n_q - (k-r) - i - 1)) \\ &+ (k-1)T(n_1 - 1) + \mathcal{O}(p(n)) \end{aligned}$$

First, a simple lower bound on the complexity of the algorithm can be derived by the fact that the procedures `RIGHT_MERGE` and `LEFT_MERGE` provide (among others) two subproblems of size $(n-1)$, based on which the following inequality holds:

$$T(n) > 2T(n-1) \tag{3.7}$$

By solving the recurrence, we obtain that $T(n) = \omega(2^n)$. Here the $\omega()$ notation is adopted to express an asymptotic lower bound of the complexity. $f(x) = \omega(g(x))$ if and only if for any positive constant c, $\exists n'$ such that $\forall x > n'$, $f(x) > cg(x) \geq 0$. As a consequence, the following inequality holds:

$$T(n) > T(n-1) + \ldots + T(1) \tag{3.8}$$

In fact, if it does not hold, we have a contradiction on the fact $T(n) = \omega(2^n)$. Now, we consider the summation $\sum_{i=k}^{n_1-(k-r)-2}(T(n_q - (k-r) - i - 1))$. Since $n_q = n_1 + q - 1$, we

can simply expand the summation as follows:

$$\sum_{i=k}^{n_1-(k-r)-2} (T(n_q-(k-r)-i-1)) = T(q)+...+T(n_1-(k-r)+q-k-2)$$

. We know that $k \geq q$, then $q-k \leq 0$ and the following inequality holds:

$$T(q)+...+T(n_1-(k-r)+q-k-2) \leq \sum_{i=q}^{n_1-(k-r)-2} T(i)$$

.

As a consequence, we can bound above $T(n)$ as follows:

$$\begin{aligned}
T(n) =& 2T(n-1)+2T(n-k-1)+...+2T(k) \\
&+\sum_{r=2}^{k}\sum_{q=1}^{r-1}\sum_{i=k}^{n_1-(k-r)-2} (T(i)+T(n_q-(k-r)-i-1)) \\
&\leq 2T(n-1)+2T(n-k-1)+...+2T(k) \\
&\quad +\sum_{r=2}^{k}\sum_{q=1}^{r-1}\sum_{i=q}^{n_1-(k-r)-2} 2T(i)+(k-1)T(n_1-1)+\mathcal{O}(p(n)) \\
&\leq 2T(n-1)+2T(n-k-1)+...+2T(k) \\
&\quad +\sum_{r=2}^{k}\sum_{q=1}^{r-1}\sum_{i=1}^{n_1-(k-r)-2} 2T(i)+(k-1)T(n_1-1)+\mathcal{O}(p(n))
\end{aligned}$$

By using Equation 3.8, we obtain the following:

$$\begin{aligned}
T(n) &\leq 2T(n-1)+2T(n-k-1)+...+2T(k) \\
&\quad +\sum_{r=2}^{k}\sum_{q=1}^{r-1}\sum_{i=1}^{n_1-(k-r)-2} 2T(i)+(k-1)T(n_1-1)+\mathcal{O}(p(n)) \\
&\leq 2T(n-1)+2T(n-k-1)+...+2T(k) \\
&\quad +\sum_{r=2}^{k}\sum_{q=1}^{r-1} 2T(n_1-(k-r)-1)+(k-1)T(n_1-1)+\mathcal{O}(p(n))
\end{aligned}$$

Finally, we apply some algebraic steps and we use the equality $n_1 = n-k$ to derive the

following upper limitation of $T(n)$:

$$\begin{aligned}
T(n) &\leq 2T(n-1) + 2T(n-k-1) + ... + 2T(k) \\
&\quad + \sum_{r=2}^{k}(r-1)2T(n_1-(k-r)-1) + (k-1)T(n_1-1) + \mathcal{O}(p(n)) \\
&\leq 2T(n-1) + 2T(n-k-1) + ... + 2T(k) + 2(k-1)T(n_1-1) \\
&\quad + \sum_{r=2}^{k-1}(r-1)2T(n_1-(k-r)-1) + (k-1)T(n_1-1) + \mathcal{O}(p(n)) \\
&\leq 2T(n-1) + 2T(n-k-1) + ... + 2T(k) \\
&\quad + (k-1)4T(n_1-1) + (k-1)T(n_1-1) + \mathcal{O}(p(n)) \\
&\leq 2T(n-1) + 4T(n-k-1) + 5(k-1)T(n-k-1) + \mathcal{O}(p(n)) \\
&= 2T(n-1) + (5k-1)T(n-k-1) + \mathcal{O}(p(n))
\end{aligned}$$

Note that $\mathcal{O}(p(n))$ includes the cost for creating all nodes for each level and the cost of all the merging operations, performed in constant time.

The recursion $T(n) = 2T(n-1)+(5k-1)T(n-k-1)+\mathcal{O}(p(n))$ is an upper limitation on the running time of TTBM. Recall that its solution is $T(n) = \mathcal{O}^*(c^n)$ where c is the largest root of the function:

$$f_k(x) = 1 - \frac{2}{x} - \frac{5k-1}{x^{k+1}} \tag{3.9}$$

.

As k increases, the function $f_k(x)$ converges to $1 - \frac{2}{x}$, which induces a complexity of $\mathcal{O}^*(2^n)$. Table 3.3 shows the time complexity of TTBM obtained by solving Equation 3.9 for all the values of k from 3 to 20.[2] The base of the exponential is computed by solving Equation 3.9 by means of a mathematical solver and rounding up the fourth digit of the solution. The table shows that the time complexity is $\mathcal{O}^*(2.0001^n)$ for $k \geq 20$. □

Lemma 15. *The problem* $1||\sum T_i$ *can be solved in* $\mathcal{O}^*((2+\epsilon)^n)$ *time and polynomial space, where* $\epsilon > 0$ *can be arbitrarily small.*

Proof. Lemma 12 and lemma 14 proved that $LPT = EDD$ is the worst-case scenario for left merging and right merging. Since $k \leq \frac{n}{2}$, the time reduction obtained from left merging and right merging, when both are incorporated into TTBM, can be combined. Thus, lemma 12 and lemma 14 together prove that instances with $LPT = EDD$ are the worst-case instances for TTBM. Therefore, the current lemma is proved according to Proposition 8. □

[2]SageMath (`http://www.sagemath.org`) is used for the computation.

k	$T(n)$
3	$\mathcal{O}^*(2.5814^n)$
4	$\mathcal{O}^*(2.4302^n)$
5	$\mathcal{O}^*(2.3065^n)$
6	$\mathcal{O}^*(2.2129^n)$
7	$\mathcal{O}^*(2.1441^n)$
8	$\mathcal{O}^*(2.0945^n)$
9	$\mathcal{O}^*(2.0600^n)$
10	$\mathcal{O}^*(2.0367^n)$
11	$\mathcal{O}^*(2.0217^n)$
12	$\mathcal{O}^*(2.0125^n)$
13	$\mathcal{O}^*(2.0070^n)$
14	$\mathcal{O}^*(2.0039^n)$
15	$\mathcal{O}^*(2.0022^n)$
16	$\mathcal{O}^*(2.0012^n)$
17	$\mathcal{O}^*(2.0007^n)$
18	$\mathcal{O}^*(2.0004^n)$
19	$\mathcal{O}^*(2.0002^n)$
20	$\mathcal{O}^*(2.0001^n)$

Table 3.3: The time complexity of TTBM for values of k from 3 to 20

3.4 Experimental results

The whole mechanism of *Branch & Merge* has been implemented and tested on instances generated in the same way as in Szwarc et al. (2001). 200 instances are generated randomly for each problem size using the same generation scheme as in the paper of Szwarc et al. (2001). Processing times are integers generated from an uniform distribution in the range $[1, 100]$ and due dates d_i are integers from a uniform distribution in the range $[p_i u, p_i v]$ where $u = 1 - T - R/2$ and $v = 1 - T + R/2$. Each due date is set to zero whenever its generated value is negative. Twenty combinations (R, T) are considered where $R \in \{0.2, 0.4, 0.6, 0.8, 1\}$, and $T \in \{0.2, 0.4, 0.6, 0.8\}$. Ten instances are generated for each combination and the combination $(R = 0.2, T = 0.6)$ yields the hardest instances as reported in the literature (see Szwarc et al. (1999)) and confirmed by our experiments. The tests are performed on a HP Z400 work station with 3.07GHz CPU and 8GB RAM.

Before comparing *Branch & Merge* to the state-of-the-art algorithm in Szwarc et al. (2001), we first describe the latter one more clearly. That algorithm, named BR-SPM, is based on the branching structure of BR, with the following three extra features integrated.

1. *Split*, which decomposes a problem according to precedence relations;
2. *PosElim*, which eliminates non-promising branching positions before each branching;
3. *Memorization*, which avoids solving a subproblem more than once by saving its solution to a database (the "memory") and retrieve it whenever the same subproblem appears again.

We implemented BR-SPM in the following way: on a given problem P, BR-SPM first tries to find the solution in the *memory*; if failed, it applies *Split* to decompose P into subproblems then solve them recursively; if *Split* does not decompose, *PosElim* is called which returns the list of positions on which the longest job can be branched on. The branching then occurs and the resulting subproblems are then solved recursively. Each time we solve a subproblem by branching, the solution sequence is saved to the memory for further query. Our implementation of BR-SPM successfully solves instances with up to 900 jobs in size, knowing that the original program was limited to instances with up to 500 jobs due to memory size limit.

We now provide our experimental results. In order to verify the concept of *merging* without extra features, we first compare *Branch & Reduce* with *Branch & Merge* on the hardest subset of instances of each size. Table 3.4, depicting (minimum, average and maximum) CPU time, average number of merging hits and total number of explored nodes, shows that the merge mechanism strongly accelerates the solution. However *Branch & Merge* is still limited to 50 jobs in size only.

	TMin	TAvg	TMax	#Merge	#Nodes
BR	52.0	1039.0	3127.0	0	1094033204
BM	3.0	67.6	319.0	11277311	47143367

Table 3.4: Results for instances of size 40

To improve the performances, we enable *Split* and *PosElim*. The resulting algorithms are called BR-SP and BM-SP. Now both algorithms can handle instances with up to 300 jobs (see Table 3.5). Surprisingly, however, even with a considerable number of merged nodes, BM-SP turns out to be slower than BR-SP.

	TMin	TAvg	TMax	#Merge	#Nodes
BR-SP	504.0	3000.8	7580.0	0	634569859
BM-SP	521.0	3097.9	7730.0	608986	508710322

Table 3.5: Results for instances of size 300

Auxiliary tests show that *Split* and *PosElim* negatively affect the merge mechanism. Solving a small problem by *Split*, which sometimes finds directly the solution sequence according to precedence relations, may be faster than merging two nodes. *PosElim* is also powerful as the average number of branching positions at each node after its application is approximately 2, i.e. most positions are already eliminated before merging. This implies that the search tree explored by BR-SP may be even smaller than a binary tree, as there are also many nodes that are not counted: they only have a single child node. These observations show that it is not straightforward to combine directly the current merge scheme with existing solving techniques. The theoretical effectiveness and running time guarantee of merging stays valid, however we need to find a new way to apply it in practice. This is more developed in chapter 4.

3.5 Additional Results

In this section, two additional results are described. The first one is the complexity analysis of the algorithm of Szwarc et al. (2001). The second one is TTBM with right-merge removed, but with Property 4 considered.

3.5.1 The complexity of the algorithm of Szwarc et al. (2001)

The current state-of-the-art algorithm described by Szwarc et al. (2001), noted hereafter as BB2001, is a branch and bound algorithm having a similar structure as that of TTBR1. The main difference is that in BB2001, in addition to the decomposition rule given in Property 2 (as in TTBR1), another decomposition rule, based on Property 5, is applied simultaneously when branching on a node. We provide in Proposition 9 the time complexity analysis of BB2001.

Property 5. *(Della Croce et al., 1998) Let job k in LPT sequence correspond to job* $[1]$ *in EDD sequence. Then, job k can be set only in positions $h \leq (n-k+1)$ and the jobs preceding job k are uniquely determined as $B_k(h)$, where $B_k(h) \subseteq \{k+1, k+2, \ldots, n\}$ and $\forall i \in B_k(h), j \in \{n, n-1, \ldots, k+1\} \setminus B_k(h)$, $d_i \leq d_j$*

Proposition 9. *Algorithm BB2001 runs in $\mathcal{O}^*(2.4143^n)$ time and polynomial space in the worst case.*

Proof. Before branching on a node, BB2001 first computes the possible positions for the longest job and the job with smallest due date. Then a new branch is created by assigning a pair of compatible positions to these two jobs. We consider two cases as follows.

Firstly, consider the case where job $1 = [n]$. The two decomposition rules become identical and if this condition is also verified in all subproblems, then the time complexity is $\mathcal{O}^*(2.4143^n)$ as proved in Proposition 2.

In the case where $1 \neq [n]$, the worst case occurs when $1 = [2]$ and $[n] = 2$, since in this case we have a maximum number of available branching positions: job $[n]$ can be branched in position $i \in \{1, ..., n-1\}$ and job 1 can be branched in position $j \in \{2, ..., n\}$, with $i < j$ for each branching. Moreover, we recall that Property 4 remains valid.

Three subproblems (left, middle and right) are created on each double branching (zero-sized problems are counted). For the sake of simplicity, we note $T(l, m, r) = T(l) + T(m) + T(r)$. The following recurrence holds.

$$T(n) = \sum_{\substack{i=1 \\ i \text{ is odd}}}^{n-1} \sum_{\substack{j=i+1 \\ j \text{ is even}}}^{n} (T(i-1, j-i-1, n-j)) + \mathcal{O}(p(n)) \tag{3.10}$$

$$= T(0,0,n-2) + T(0,2,n-4) + T(0,4,n-6) + ... + T(0,n-2,0) + \tag{3.11}$$

$$T(2,0,n-4) + T(2,2,n-6) + ... + T(2,n-4,0) + \tag{3.12}$$

$$... \tag{3.13}$$

$$T(n-4,0,2) + T(n-4,2,0) + \tag{3.14}$$

$$T(n-2,0,0) + \tag{3.15}$$

$$\mathcal{O}(p(n)) \tag{3.16}$$

$$= 3 * (T(n-2) + 2T(n-4) + 4T(n-6) + ... + \frac{n}{2}T(0)) + \mathcal{O}(p(n)) \tag{3.17}$$

$$\tag{3.18}$$

By applying a similar process of simplification as in the proof of Proposition 2, the following result is finally derived:

$$T(n) = 5T(n-2) - T(n-4). \tag{3.19}$$

Correspondingly, we have $T(n) = \mathcal{O}^*(\sqrt{\frac{5+\sqrt{21}}{2}}^n) = \mathcal{O}^*(2.1890^n)$. Therefore the worst case occurs when the two decomposition rules overlap, and the resulting time complexity is the same as TTBR2, namely $\mathcal{O}^*(2.4143^n)$. The space complexity of BB2001 is also polynomial since depth-first exploration is adopted.

□

3.5.2 Algorithm TTBM-L

An intermediate algorithm has been derived based on TTBR1, with Left-Merge enabled, Right-Merge disabled, and Property 4 included. The algorithm is named as TTBM-L and it is stated in Algorithm 6. This is basically the algorithm applied on the example problem in section 3.3.1.1 without considering Properties 3 and 4.

Each time a node is opened, the sub-branches required for left-merging operations are generated, the subproblems of size less than k are solved and the procedure `LEFT_MERGE` is called. Then, the algorithm proceeds recursively by extracting the next node from Q with a depth-first strategy and terminates when Q is empty.

Proposition 10. *Algorithm TTBM-L runs in $\mathcal{O}^*((2.247+\epsilon)^n)$ time and polynomial space, where $\epsilon \to 0$ when $k \to \infty$.*

Proof. Starting from Algorithm 6, we can derive that for a given problem P of size n, the $(k-1)$ first children nodes P_2 to P_k are merged with children nodes of P_1. Consequently, among these nodes, only node P_1 remains as a child node of P. For the other $(n-k)$ children nodes, Property 3 is applied eliminating by the way one node over two. The worst-case is achieved when n is odd and k is even and we have the following recurrence:

$$T(n) = T(n-1) + (T(n-k-1) + T(k)) + (T(n-k-3) + T(k+2)) + ...$$

Algorithm 6 Total Tardiness Branch and Merge (Left Only) (TTBM-L)

Input: $P : \{1, ..., n\}$: input problem of size n
$k \geq 2$: an integer constant
Output: *seqOpt*: an optimal sequence of jobs

```
1: function TTBM-L(P,k)
2:     Q ← P
3:     seqOpt ← a random sequence of jobs
4:     while Q ≠ ∅ do
5:         P* ← extract next problem from Q (depth-first order)
6:         if (the size of P* < k) then Solve P* by calling TTBR1
7:         end if
8:         if all jobs {1, ..., n} are fixed in P* then
9:             seqCurrent ← the solution defined by P*
10:            seqOpt ← best solution between seqOpt and seqCurrent
11:        else
12:            Q ← Q ∪ LEFT_MERGE(P*)
13:            for i = k + 1, ..., n do
14:                Create child node P_i like in TTBR1
15:                if P_i is not eliminated by Property 3 then Q ← Q ∪ P_i
16:                end if
17:            end for
18:        end if
19:    end while
20:    return seqOpt
21: end function
```

$$+(T(2) + T(n-3)) + T(n-1) + \mathcal{O}(p(n))$$

which can be reformulated as

$$T(n) = 2T(n-1) + T(n-3) + ... + T(n-k+1) + 2T(n-k-1) + ... + 2T(2) + \mathcal{O}(p(n))$$

Following the same approach used in the proof of Proposition 2, we plug $T(n-2)$ into the formula and we have

$$T(n) = 2T(n-1) + T(n-2) - T(n-3) + T(n-k-1) + \mathcal{O}(p(n)) - \mathcal{O}(p(n-2))$$

The solution of this recurrence is $T(n) = \mathcal{O}^*(c^n)$ with c the largest root of

$$1 = \frac{2}{x} + \frac{1}{x^2} - \frac{1}{x^3} + \frac{1}{x^{k+1}}$$

When k is large enough, the last term in the equation can be ignored, leading to a value of c which tends towards 2.247 as k increases. □

3.6 Chapter summary

This chapter focuses on the design of exact branching algorithms for the single machine total tardiness problem. By exploiting some inherent properties of the problem, we first proposed two *Branch & Reduce* algorithms: TTBR1 and TTBR2. The former runs in $\mathcal{O}^*(3^n)$ time, while the latter achieves a better time complexity in $\mathcal{O}^*(2.4143^n)$. The

space requirement is polynomial in both cases. Furthermore, a technique called merging, is presented and applied onto TTBR1 in order to improve its performance. The final achievement is a new algorithm (TTBM) with time complexity converging towards $\mathcal{O}^*(2^n)$ and requiring polynomial space. The same technique can be tediously to improve the performance of TTBR2, but the resulting algorithm achieves the same asymptotic time complexity as TTBM, and thus it was omitted. To the best of authors' knowledge, TTBM is the polynomial space algorithm that has the best worst-case time complexity for solving the $1||\sum T_j$ problem.

Beyond the new established complexity results, the main contribution of the paper is the *Branch & Merge* algorithm. The basic idea is very simple, and it consists of speeding up branching algorithms by avoiding to solve identical problems. The same goal is traditionally pursued by means of *Memorization* (Fomin and Kratsch, 2010), where the solution of already solved subproblems are stored and then queried when an identical subproblem appears. This is at the cost of an exponential space requirement. In contrast, *Branch & Merge* also discards identical subproblems but by appropriately merging, in polynomial time and space, nodes involving the solution of common subproblems. When applied systematically in the search tree, this technique enables to achieve a good worst-case time bound. On the computational side, it should be noticed that the merging operation can be relaxed to obtain more efficiency in practice. Instead of comparing nodes at specific positions and solving in $\mathcal{O}^*(2.4143^k)$ subproblems at merged nodes, as described in *Branch & Merge*, we may just compare active nodes with already branched nodes and cut the dominated ones, keeping a polynomial space usage. This can also be seen as memorization but with a fixed size memory used to store already explored nodes. This leads to a weaker worst-case time bound but early works of T'kindt et al. (2004) have shown that this can lead to substantially good practical results, at least on some scheduling problems.

As a further development of this work, our aim is twofold. First, we aim at applying the *Branch & Merge* algorithm to other combinatorial optimization problems in order to establish its potential generalization to other problems. Second, we want to explore further the practical efficiency of this algorithm on the single machine total tardiness problem, with a different implementation, in a similar way as done by Szwarc et al. (2001) and T'kindt et al. (2004). The first aim has not yet been achieved at this point (the end of this thesis) because a direct application of the *Branch & Merge* algorithm requires the target problem to verify certain properties like Property 2. However, much work has been performed on the second direction, which is about the practical efficiency of the algorithm. The Chapter 4 is dedicated to this topic.

The main work in this chapter has been performed together with Michele Garraffa, Federico Della Croce from *Politechnico di Torino* (Italy) and Vincent T'Kindt. The *Branch & Reduce* algorithms TTBR1 and TTBR2 have been reported in the *MISTA* 2015 conference (Della Croce et al., 2015b) and an Italian conference *AIRO* 2015 (Della Croce et al., 2015a). The main results on *Branch & Merge* have been reported at the international conference on *Project Management and Scheduling* (PMS 2016) (Shang et al., 2016a) and the French conference *ROADEF* 2017 (Shang et al., 2017b). The additional results have been reported at the *International Symposium on Parameterized and Exact Computation* (IPEC 2017) (Shang et al., 2017a). A journal paper submitted to *Theoretical Computer Science* is currently under review and the paper can be accessed on HAL (Garraffa et al.,

2017).

Chapter 4

The Memorization Paradigm: *Branch & Memorize* Algorithms for the Efficient Solution of Sequencing Problems

4.1 Introduction

The algorithm *Branch & Merge* presented in chapter 3 is proved to be not efficient enough in practice. We therefore turned our interest to another technique called *Memorization*, which was adopted by Szwarc et al. (2001) to efficiently solve the $1||\sum T_i$ problem. It was also used, though not in a typical way, by T'kindt et al. (2004) to solve three sequencing problems, notably the $1|r_i|\sum C_i$, the $1|\tilde{d}_i|\sum w_i C_i$ and the $F2||\sum C_i$ problems. In this chapter we study, experiment, compare and analyze the efficiency of this method on the four above mentioned problems.

Memorization as an algorithm design technique, allows to speed up algorithms at the price of more space usage. Typically in branching algorithms, on lower branching levels, isomorphic subproblems may appear exponentially many times and the idea of Memorization is to avoid repetitive solutions as they correspond to identical subproblems. The method was first applied on the *Maximum Independent Set* problem by Robson (1986). By exploiting graph theoretic properties and by applying *Memorization* to avoid solving identical subproblems, Robson proposed an algorithm with a worst-case time complexity in $\mathcal{O}(1.2109^n)$. It has remained the exact exponential algorithm with the smallest worst-case time complexity until 2013, when it was improved by the $\mathcal{O}(1.1996^n)$ algorithm of Xiao and Nagamochi (2017). *Memorization* is sometimes used to speed up branching algorithms (Chandran and Grandoni, 2005; Fomin et al., 2005b; Fomin and Kratsch, 2010) in the context of EEA (*Exact Exponential Algorithms*), where the objective is to conceive exact algorithms that can provide a best possible worst-case running time guarantee.

Despite of the fact that a typical *Memorization* algorithm memorizes solutions of subproblems that appear repeatedly, we prefer to interpret the idea in a more general way.

What we call the *Memorization Paradigm* can be formulated as "Memorize and learn from what have been done so far, to improve the next decisions". In the literature, various algorithms can be classified as procedures embedding memorization techniques, though the implementation could be quite different depending on the problem structure and the information to store. For instance, *Tabu Search* (Glover, 1989, 1990) is a metaheuristic, which memorizes recently visited solutions in order to avoid returning back to these solutions again during the search. SAT solvers deduce and then memorize conflict clauses during the tree search in order to perform non-chronological backtracking (*Conflict Driven Clause Learning*) (Biere et al., 2009; Zhang et al., 2001). Similar ideas also appear in *Artificial Intelligence* area as *Intelligent Backtracking* or *Intelligent Back-jumping*.

From a theoretical point of view, the drawback relies on the memory consumption of *Memorization* which can be exponential. This drawback turns out to limit the quantity of memorized information like in *Tabu Search* or SAT solvers. In this chapter we instantiate the *Memorization Paradigm* in a way similar to what is done in the field of EEA, i.e. we set up a *Memorization* framework for search tree based exact algorithms but with a control on the memory usage. We had the intuition that a *Memorization* with limited memory could already dramatically accelerate the solution in practice. By embedding a simple *Memorization* technique into their *Branch & Bound* algorithm, Szwarc et al. (2001) solve the single machine total tardiness problem on instances with up to 500 jobs in size. Other works presenting standard memorization techniques applied to sequencing problems have been published by T'kindt et al. (2004) where the benefit of such technique is well shown.

As introduced in section 1.2.2.1, branching algorithms are based on the idea of enumerating all possibilities via a search tree created by a branching mechanism. For each decision variable, the algorithm *branches* on all possible values, each time creating a new subproblem (a node in the search tree) of a reduced size. The algorithm continues recursively and returns the global optimal solution. The basic structure being simple, the critical question is how to prune the search tree so as to avoid exploring unpromising nodes. Dominance conditions are commonly used to cut nodes: at a node, if it is proved that a more promising node exists or can be easily found, then the current one can be abandoned. This is also the case for *Branch & Bound*, in which at each node, the bounding procedure provides an optimistic estimation of the solution quality of that node. If the estimation value is not better than the currently best solution found, in other words, the current node is dominated by the incumbent solution, then the node is cut without being further developed. Just like the bounding procedure in *Branch & Bound*, *Memorization* can be seen as another procedure which can help in pruning the search tree. In branching algorithms, especially on lower branching levels, isomorphic subproblems may appear exponentially many times and *Memorization* can be used to avoid solving identical problems multiple times.

Memorization, apparently, has not yet been systematically considered when designing search tree based algorithms, as the bounding procedure in *Branch & Bound*. It is at least rare in sequencing problems, to the authors' knowledge. The aim of this work is to promote a systematic integration of *Memorization* into search tree based algorithms in order to better prune the search tree. In the following sections, we first describe a general framework of *Memorization* (section 4.2), followed by some guidelines on the implementation (section 4.3). Then, we apply the framework to four scheduling problems including $1|r_i|\sum C_i$ (section 4.4.1), $1|\tilde{d}_i|\sum w_i C_i$ (section 4.4.2), $F2||\sum C_i$ (section 4.4.3) and $1||\sum T_i$ (section

4.5). Finally, we conclude this work in section 4.6.

4.2 A general framework for *Memorization* in search trees

For a given minimization problem, the application of *Memorization* depends on several components of the search tree based algorithm such as the branching scheme, the search strategies and also the characteristics of the problem. In this section, we consider possible scenarios that may appear for sequencing problems. Then, we present the possible schemes of *Memorization* and how to choose the right scheme depending on the scenario.

Even though the general idea of *Memorization* can be generalized and applied to any combinatorial optimization problems, we focus on sequencing problems in the context of the thesis. Consider a generic sequencing problem where n jobs $J = \{1, ..., n\}$ are to be scheduled. Each job i is defined by a set of features like a processing time p_i, a due date d_i, etc, which depends on the problem under consideration. Some resources are available for the execution of jobs and an ordering of jobs must be found to minimize some cost function, usually depending on the jobs completion times. We adopt an intuitive way to represent the content of a node or a subproblem: as an example $123\{4, ..., n\}$ represents a subproblem in which jobs $\{1, 2, 3\}$ are already fixed by branching, to the first three positions of the sequence, while the jobs to be scheduled after are $\{4, ..., n\}$.

No matter the branching scheme, at any iteration of the algorithm, by *active nodes* we denote the nodes that are created but not yet developed, and by *explored nodes* the nodes that have already been branched on (children nodes have been created). We also adopt the notion of *decomposable problems* defined by T'kindt et al. (2004). Typically for single machine scheduling problems, this often implies that the completion time of the prefixed job sequence of a node is constant no matter of the order of jobs inside (it is defined as the sum of processing times of the jobs in that sequence).

Definition 7. *Let $\{1, ..., i\}\{i+1, ..., n\}$ be a problem to be solved. It is decomposable if and only if the optimal solution of the subproblem $\{1, ..., i\}$ (resp. $\{i+1, ..., n\}$) can be computed independently from $\{i+1, ..., n\}$ (resp. $\{1, ..., i\}$), i.e. without knowing the optimal sequence of $\{i + 1, ..., n\}$ (resp. $\{1, ..., i\}$).*

4.2.1 Branching schemes

In common search tree based algorithms for scheduling (sequencing) problems, the branching operation consists in assigning a job to a specific position in the sequence. A *Branching Scheme* defines, at a node, how to choose this job and the positions to occupy. We consider three classic branching schemes, namely *forward branching*, *backward branching* and *decomposition branching*.

When applying *forward branching* at a given node, each eligible free job is assigned to the first free position. For example, the nodes at the first level of the search tree correspond to the following subproblems: $1\{2, ..., n\}$, $2\{1, 3, ..., n\}$, ..., $n\{1, ..., n-1\}$.

When applying *backward branching* at a given node, each eligible free job is assigned to the last free position. For example the nodes at the first level of the search tree correspond

to the following subproblems: $\{2, ..., n\}1$, $\{1, 3, ..., n\}2$, ..., $\{1, ..., n-1\}n$. This scheme is symmetric with *forward branching*, hence for the sake of simplicity we only discuss *forward branching* and add extra remarks on *backward branching* whenever necessary.

When applying *decomposition branching* at a given node, the job that is being considered to branch is called a *decomposition job*. When a *decomposition job* is assigned to a position, two subproblems are generated implied by the free positions before and after the *decomposition job*. Certainly one may determine the jobs that should be scheduled before and after this position by enumerating all 2-partitions of jobs as the *Divide & Conquer* technique introduced by Fomin and Kratsch (2010), but here we restrict our study to the situation where the two subproblems can be uniquely determined in polynomial time making use of some specific problem properties. As an example, the nodes at the first level of the search tree could contain $\{2, 3, 4\}1\{5, ..., n\}$, if job 1 is the *decomposition job* which is assigned to position 4 and generates two subproblems corresponding to jobsets $\{2, 3, 4\}$ and $\{5, ..., n\}$, respectively. This situation occurs, for instance, to the $1||\sum T_i$ problem as already introduced in chapter 3.

4.2.2 Search strategies

During the execution of search tree based algorithms, when two or more nodes are active, a strategy is needed to determine the next node to branch on. The classic search strategies are *depth first*, *best first* and *breadth first*.

Depth first is the most common strategy: the node to explore is an arbitrary active node at the lowest search tree level. The advantage of this strategy is that it only requires polynomial space.

Breadth first selects an active node with the highest search tree level. This leads to an exponential space usage since the search tree is explored level by level.

Best first chooses the node to explore according to its lower bound. The space usage in the worst case is therefore also super-polynomial like in *breadth first*.

It seems conventional that when constructing a search tree based algorithm, the *depth first* is adopted. However, this choice is strongly questionable according to T'kindt et al. (2004).

4.2.3 Memorization schemes

The *memorization* presented by Robson (1986) stores the optimal solution of each subproblem of a predetermined limited size and reuses that solution whenever such subproblem appears again during the tree search. However different memorization approaches can be used. The differences rely on the choice of the information to store and the way in which the stored information is used. We discuss below three different memorization schemes that are helpful to efficiently solve some sequencing problems.

Taking account of the branching schemes introduced in section 4.2.1, any node of the search tree can be defined by $\sigma_1 S_1 \sigma_2 S_2 ... \sigma_k S_k$, the $\sigma_j{}'s$ being partial sequences of jobs and the $S_j{}'s$ being subproblems which remain to be scheduled. For the sake of simplicity, we explain the *Memorization* schemes in the case of *forward branching*, i.e. $k = 1$, and a node

corresponds to a problem σS.

4.2.3.1 Solution memorization

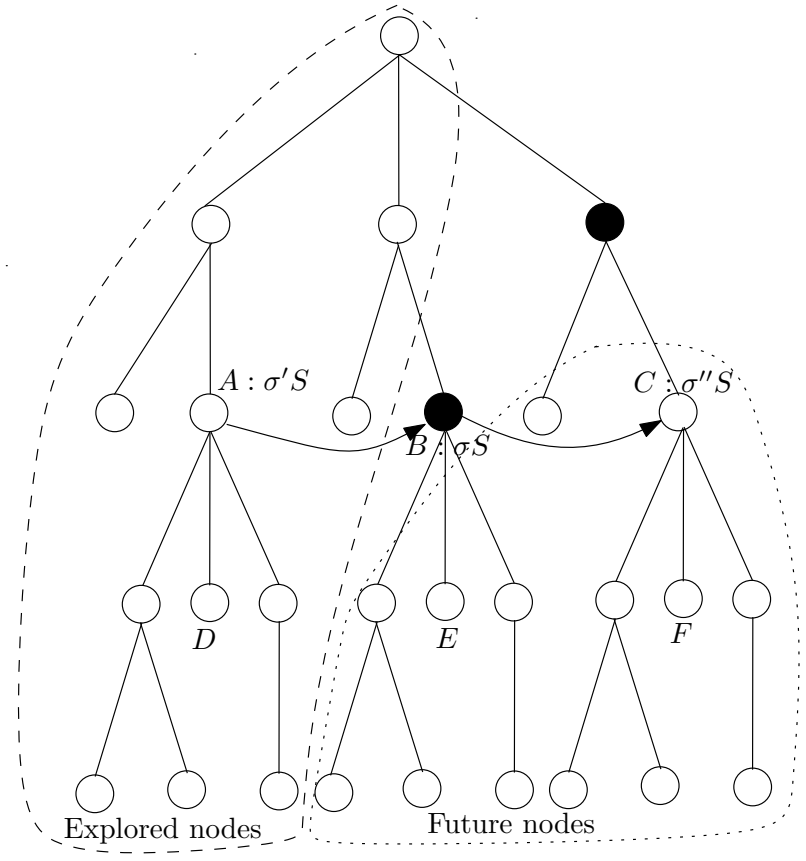


Figure 4.1: Solution Memorization

Consider the situation illustrated in Figure 4.1, where active nodes are colored in black. Node B is the current node, while σ, σ' and σ'' are different permutations of the same jobset. In other words, nodes A, B and C may contain the same subproblem to solve, implied by S. In that case, if A has already been solved (consider for instance a depth-first search) and the optimal sequence of S has been memorized, then it can be used directly to solve nodes B and C and it is no longer necessary to branch on these nodes. Note that, in order to successfully perform memorization, we must guarantee that the solution of S memorized at node A is optimal. Depending on the branching algorithm implementation, this may not be obvious: for instance in *Branch & Bound* algorithms, the leaf node corresponding to the optimal solution of node A may be missed if one of its ascendant node is cut due to a dominance condition. Looking at Figure 4.1, assume that node D should have led to the optimal solution of problem S but has been cut by a dominance condition. Applying *solution memorization* may then lead to memorize another solution β to S, which is not optimal with respect to S. Troubles may appear if the global optimal solution to the original problem (associated to the root node) is, for instance, given by node E. Solution

memorization may imply not exploring node B and directly replacing S by the "best" solution found from node A. As a consequence, the global optimal solution is missed. This situation occurs whenever the dominance condition which has pruned node D would not have pruned node E: in the remainder, this kind of conditions are refined to as *context dependent dominance conditions* since they depend on the context of each node (typically, the initial partial sequence σ, σ' and σ''). By opposition, a *context independent dominance condition* would have pruned node D, E and F. A direct way to fix this is to disable dominance conditions whenever *solution memorization* is applied.

However, if these context dependent conditions are playing a very important role in the algorithm then this may slow down the algorithm even if *solution memorization* works. Another approach to manage context dependent dominance conditions is to extend the memorization from "solutions" to "lower bounds" when the branching algorithm involves a bounding mechanism. In that version of *Memorization*, we assume that all dominance conditions are kept in the algorithm. When node A is created, a lower bound is computed, which represents the best solution value we may expect from the subtree of A. This lower bound is based on the cost function value of the sequence σ' which is already fixed, and an evaluation on the unsolved part S. When branching down the subtree of A, jobs in S are fixed gradually, hence the evaluation on the remaining unscheduled jobs also becomes more and more precise. When all leaf nodes of the subtree of A are explored, this value finally becomes tighter (higher) than the initial value computed at node A. Since the objective function value of σ' is known, we can then deduce the lower bound value corresponding to S when scheduled after σ', and memorize it. Now when node B is opened, instead of computing its lower bound, we can get it by finding the lower bound of S directly from the memory and then add the objective function value of σ. In this way, the lower bound we get is tighter, and node B is more likely to be cut. Moreover, the lower bound computation at node B, which may be time costly, is saved. Notice that, for nodes cut by context dependent dominance conditions, their lower bound values still need to be computed and considered (hence introduces an extra cost). Lower bound memorization can be a good alternative to *solution memorization* with context dependent dominance conditions turned off as long as these conditions are efficient in pruning the search tree.

Note that the memorization of lower bounds is compatible with the memorization of optimal solutions: whenever in a subtree no nodes are cut by context dependent dominance conditions and the global upper bound is updated by some nodes from this subtree, the optimal solution of this subtree is memorized. Otherwise, the lower bound is memorized. We denote the described memorization technique including the memorization of optimal solutions and the memorization of lower bounds as *solution memorization* since both are related to the memorization of the "best solution" of the problem associated to a node.

4.2.3.2 Passive node memorization

At any node σS, another information that can be memorized is the partial sequence σ. Unlike *solution memorization* where the memorized sequences can be used to solve a node, *passive node memorization* is only used to cut nodes.

Consider the branching situation depicted by Figure 4.2. Again active nodes are black-colored and B is the current node. Assume a node A exists among explored node, with

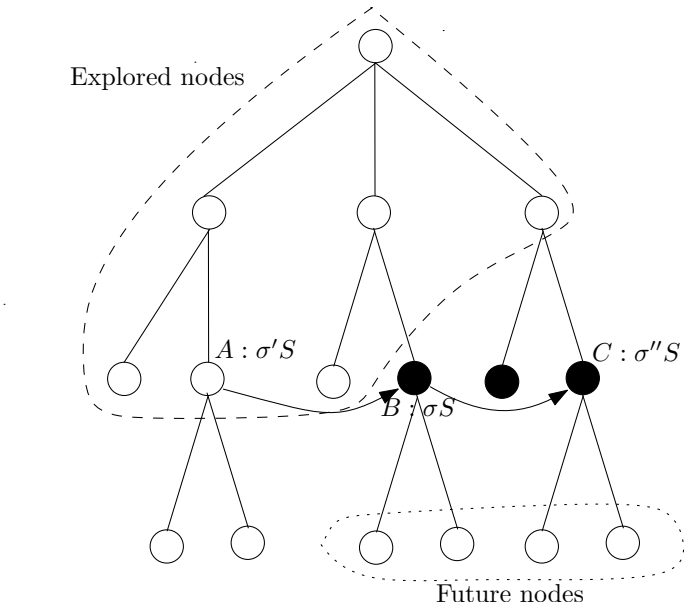


Figure 4.2: Passive node memorization)

σ' being a different permutation of the same jobset used in σ. If the partial sequence σ' has been memorized then two situations may occur. If σ' dominates σ then B can be cut since it cannot lead to a solution better than A. If no such σ' dominating σ is available, then σ can be memorized in order to possibly prune a future node like C. Note that *solution memorization* and *passive node memorization* may possibly intersect. Consider the previous example and nodes A, B and C. If the optimal solution of subproblem S has been obtained from the exploration of node A, then at node B and C both *solution memorization* and *passive node memorization* imply not to branch on these nodes if σ' dominates σ and σ''.

The dominance test between sequences can be implemented as a function $check(\sigma, \sigma')$ which returns `1` if σ' dominates σ, as introduced by T'kindt et al. (2004). The check must be done on two different sequences of the same jobset, having the same starting time and its implementation is problem dependent. Since the memorized sequence results from branching decisions, we call it *passive node memorization.*

Any node, ready to be branched on, must be compared to explored and/or to active nodes depending on the search strategy. Additionally, it may be necessary to perform the check twice: first once the node is created, then at the time of branching. Memorizing the partial sequence, when the node is created, ensures that the best sequence is kept before any exploration of nodes. Then, rechecking the dominance when branching on a node enables that node to be cut if a dominant partial sequence has been found meanwhile.

In the following, we introduce Property 6 which relates the lower bounding mechanism of search tree based algorithms to the *check* function. When this property is answered, the current node only needs to be compared to explored nodes instead of all nodes when best first is chosen as the search strategy, as detailed in Section 4.2.4.

Property 6. *(Concordance property) Let LB(A) be the lower bound value computed at*

node A. The search tree based algorithm satisfies the concordance property if and only if, for any node $A = \sigma S$ and $B = \pi S$, $LB(A) < LB(B) \Leftrightarrow check(\pi, \sigma) = 1$.

4.2.3.3 Predictive node memorization

Predictive node memorization relies on the same idea as *passive node memorization*, but with additional operations. As illustrated in Figure 4.3, at a given node $B = \sigma S$, we first check, like in *passive node memorization*, if the current node can be cut by σ' memorized at node A. If not, instead of directly memorizing σ, we search for an improving sequence π. Notice that, by the way, the improving sequence necessarily belongs to a part of the search tree not yet explored when dealing with the node σS. There may be many ways to compute π. For instance, we may perform some local search on σ, searching for a neighbor sequence π that dominates σ. Alternatively, we may focus on a short subsequence of σ and solve it to optimality (in a brute-force way, for instance). The latter idea appears in the work of Jouglet et al. (2004) as *Dominance Rules Relying on Scheduled Jobs*. We may also make use of an exact algorithm to optimize a part of σ to get σ', as far as this algorithm is fast. Notice that this idea is strongly related to the merging mechanism introduced in chapter 3 (see also Garraffa et al. (2017)), which is designed to provide good worst-case time complexities. If such a sequence π can be built, then the current node σS is cut and node πS is memorized. Note that node πS has not yet been encountered in the search tree

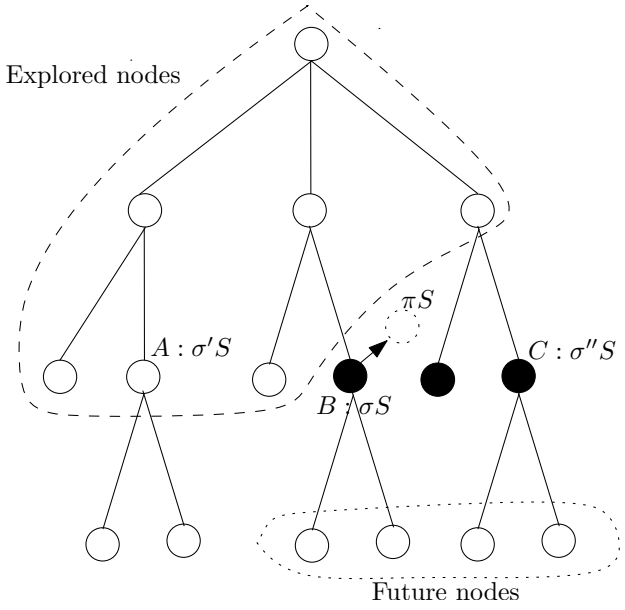


Figure 4.3: Predictive node memorization

when dealing with node σS (consider, $\pi = \sigma''$). So, it is important when applying *predictive node memorization* to remember that πS still needs to be branched on. Also, the extra cost of generating π must be limited in order to avoid excessive CPU time consumption.

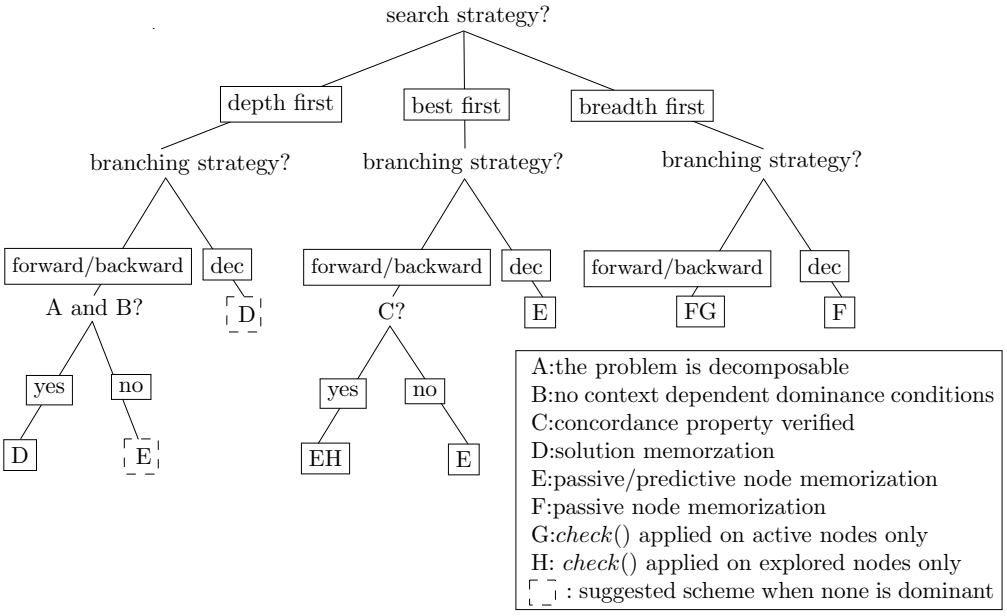


Figure 4.4: Decision tree for choosing the memorization scheme

4.2.4 Decision guidelines

In this section we provide some guidelines on how to choose the appropriate memorization scheme according to the branching scheme and the search strategy. The main results are summarized in the decision tree in Figure 4.4.

4.2.4.1 Forward branching and depth first search strategy

In *forward branching*, any node of the search tree can be defined as σS. When *depth first* is used as the search strategy we can state the following property.

Property 7. *With forward branching and depth first, if the problem is decomposable and solution memorization memorizes optimal solutions, then solution memorization dominates both passive node memorization and predictive node memorization.*

Proof. Any node deletion that can be achieved by *passive node memorization* and *predictive node memorization* can also be achieved by *solution memorization*, but not conversely. Consider nodes $A = \sigma S$ and $B = \pi S$ with σ and π two permutations of the same jobset. As the problem is decomposable, solving subproblem S at node A is equivalent to solving it at node B. Without loss of generality, we assume that A appears before B during the solution. In *passive node memorization* if $check(\pi, \sigma) = 1$, i.e. sequence σ dominates π, then B can be pruned. However, also in *solution memorization*, node B can be pruned since the optimal solution of jobset S has already been memorized from node A.

Now consider the case where $check(\sigma, \pi) = 1$, i.e. sequence π dominates σ. This implies that with *passive node memorization*, node B will not be pruned. However, as explained above, with *solution memorization*, node B is pruned. With *predictive node memorization*,

the conclusion is the same since we have no guarantee that starting from node πS another node αS dominating πS can be generated. Besides, even if such αS is generated and πS is pruned, the same issue occurs to node αS when it is generated. □

If the problem is not decomposable, or *context dependent dominance conditions* are used in the algorithm, then *solution memorization* memorizes lower bounds, and then which memorization scheme is dominant can not be determined. However, in practice, *passive node memorization* may be preferred to *solution memorization*. Notably, as the problem is not decomposable, then it may be necessary to solve the subproblem consisting of jobset S both at nodes A and B. However, with *passive node memorization*, node B may be pruned whenever π is dominated by σ.

4.2.4.2 Forward branching and best first search strategy

We can state the following property.

Property 8. *With forward branching and best first strategy, solution memorization does not apply. Passive node memorization or predictive node memorization can be applied only to explored nodes if the concordance property (Property 4.6) is answered.*

Proof. To apply *solution memorization* at a given node, the subproblem concerning S must be solved first in order to memorize its optimal solution. This is not compatible with *best first* search strategy. In fact, if *best first* reaches a leaf node, then also the optimal solution is reached and no sequence has been stored before.

When *passive node memorization* and *predictive node memorization* are applied, the search, at a given node, of a dominant subsequence needs only to be done in the set of explored nodes whenever the concordance property holds. As the *best first* search strategy always consider for branching the node with the lowest lower bound value, the concordance property implies that no active node can dominate it. □

When the concordance property does not hold, then node memorization techniques are required to consider both explored and active nodes for node pruning.

Besides, no dominance can be deduced a priori between *predictive node memorization* and *passive node memorization*. It depends on how the search for an improving subsequence is applied in *predictive node memorization*. Generally speaking, both memorization schemes should be considered and compared to find the best one.

4.2.4.3 Forward branching and breadth first search strategy

With forward branching and breadth first we can state the following property.

Property 9. *With forward branching and breadth first strategy, solution memorization does not apply. Passive node memorization should be chosen and should be applied to active nodes.*

Proof. Under this configuration, *solution memorization* is useless since leaf nodes are reached only at the end of the search tree. *Passive node memorization* can be applied to *active nodes* only. An active node A is selected for branching when all the nodes at the same level have been created, hence all other active nodes dominated by A are discarded. If in turn A it is dominated by another node, then it is pruned. There is no need to consider explored nodes since explored nodes on higher levels have less fixed jobs, therefore they are not comparable with the current node. Also, *predictive node memorization* can not do better than *passive node memorization* since *passive node memorization* already keeps the best node at each level. □

4.2.4.4 Decomposition branching and depth first search strategy

With *decomposition branching*, at each level of the search tree a decomposition job can be put on any free position by the branching operation.

Under this configuration, no dominance can be deduced among the memorization schemes. In fact, we can imagine situations where either *solution memorization* or *passive node memorization* or *predictive node memorization* is dominant. Consider nodes $A = \sigma S_1 j_1 S_2$ and $B = \pi S_1 j_2 S_3$ with A being explored before B. In both nodes, the current subproblem concerns scheduling jobset S_1 after σ or π. Suppose σ and π contain different jobs but have the same completion time, which means that the subproblem defined by S_1 is identical in A and B. Then, the optimal sequence for S_1 found when solving A can be reused on B by *solution memorization*, while *passive node memorization* cannot handle this case since σ and π contain different jobs hence, are incomparable. *Predictive node memorization* may or may not cut B depending whether a dominant prefix can be generated or not.

On the other hand, we may also imagine the case where $A = \sigma S_1 j_1 S_2$ and $B = \pi S_3 j_2 S_4$. Suppose σ and π are different permutations of the same jobset. If $check(\pi, \sigma) = 1$, then node B can be cut by *passive node memorization* or *predictive node memorization*, while this is not the case for *solution memorization* because subproblems S_1 and S_3 do not consist of the same jobs.

In practice, even though every memorization scheme could be dominant in some cases, the memory limitation does not allow to apply all of them and our experience suggests to prefer *solution memorization*. This is due to the special structure of nodes $\sigma_1 S_1 ... \sigma_k S_k$, which makes the prefixed jobs much spread out (they are separated by S_i), and prevents the application of successful *passive node memorization* and *predictive node memorization*. Moreover, the case with nodes $\sigma_1 S \sigma_2$ and $\pi_1 S \pi_2$, where σ_1 and π_1 have the same completion time but contain different jobs, may occur pretty often for large instances if the jobs processing times do not present a large variance.

4.2.4.5 Decomposition branching and best first search strategy

Property 10. *With decomposition branching and best first strategy, solution memorization does not apply. Passive node memorization and predictive node memorization must only be applied to explored nodes whenever the concordance property holds and the check function comparing two nodes* $\sigma_1 S_1 ... \sigma_k S_k$ *and* $\sigma_1' S_1' ... \sigma_{k'}' S_{k'}'$ *only works on* σ_1 *and* σ_1'*. Otherwise,*

passive node memorization and predictive node memorization must be applied to explored and active nodes.

Proof. Similar to that of Property 8. □

4.2.4.6 Decomposition branching and breadth first search strategy

Property 11. *With decomposition branching and breadth first strategy, solution memorization does not apply. Whether node memorization should be applied to active nodes only depends on the definition of the check function.*

Proof. This configuration discourages *solution memorization* for the same reason as in Property 9. If the *check* function is defined in a way such that the explored nodes are not comparable to active nodes, then *passive node memorization* and *predictive node memorization* should be applied to *active nodes* only, otherwise they should be applied to all nodes. □

4.3 Implementation guidelines

In this section we discuss efficient implementations of the memorization schemes, providing, when necessary, choices specific to the sequencing problems tackled in the remainder. The key point is to have a fast access to memorized partial solutions. Henceforth, we implement a database as a hashtable which contains all the memorized solutions. By well choosing the hash function, a hashtable supports querying in $\mathcal{O}(1)$ time to find the corresponding elements given a hash key.

For *solution memorization*, at a given node, the database is queried with $\langle t_0, S\rangle$, where t_0 is the starting time of the subproblem and S is the related jobset. The returned result should be $\langle \pi, opt(\pi|t_0)\rangle$ which is the optimal sequence associated to S when starting at time t_0, and its corresponding objective function value. So, $\langle \pi, opt(\pi|t_0)\rangle$ defines the elements which are memorized in the database. We define the hash key h as a combination of t_0 and $|S|$: seeing h as a set of bits, t_0 occupies the higher bits in h while $|S|$ occupies the lower bits. The aim is to have a unique hash key for each given pair $\langle t_0, S\rangle$, even if this is not necessarily bijective: i.e., two elements in the database with the same hash key may correspond to different pairs $\langle t_0, S\rangle$. As a consequence, when a list of elements is returned for a pair $\langle t_0, S\rangle$, it is also necessary to verify that the returned sequence is a sequence of jobset S. This takes $\mathcal{O}(|S|)$ operations for each returned sequence. We may also include the sum of job id's of S into h in order to have a more exact key, but this correspondingly increases the time needed to construct the key, without preventing from checking whether a returned sequence π is a permutation of jobset S or not.

For *passive* and *predictive node memorization*, implementation decisions are more dependent on the problem and on the check function used to compare two partial sequences

σ and π of the same jobset. For any such σ and π, a general definition of *check*() could be:

$$check(\pi,\sigma) = \begin{cases} 1, \ if \ C_{max}(\sigma) \leq \max(C_{max}(\pi); E_{min}(\pi)) \ and \ opt(\sigma|t_0) \leq opt(\pi|t_0) \\ 0, \ otherwise \end{cases} \tag{4.1}$$

with C_{max} referring to the makespan of a partial sequence, and $E_{min}(\pi)$ referring to the earliest starting time of the jobs scheduled after π. It is not difficult to see that if $check(\pi,\sigma) = 1$ then node σS dominates node πS. Indeed, for any regular objective function to minimize, with respect to the fixed jobs, $opt(\sigma|t_0) \leq opt(\pi|t_0)$ ensures that σ yields a smaller cost than π. Moreover, $C_{max}(\sigma) \leq \max(C_{max}(\pi); E_{min}(\pi))$ guarantees that the starting time of jobset S at node σS is not higher than in node πS. Therefore, σS dominates πS.

Consequently, a database element is a tuple $\langle \sigma, C_{max}(\sigma), E_{min}(\sigma), opt(\sigma|t_0), ExpAct \rangle$ with *ExpAct* being a flag indicating whether this element corresponds to an explored or an active node. Notice that t_0 is not included since it appears in the hash key used for querying. Also, when the problem is decomposable, the *check* function reduces to:

$$check(\pi,\sigma) = \begin{cases} 1, \ if \ opt(\sigma|t_0) \leq opt(\pi|t_0) \\ 0, \ otherwise \end{cases}$$

where only $\langle \sigma, opt(\sigma|t_0), ExpAct \rangle$ need to be stored. For node memorization techniques the hash key, at a given node, is computed in a way similar to solution memorization. Consider, for example, *forward branching*: let $\sigma_1 S_1$ be the current node. As the dominance of another node is checked on σ_1, the database is queried with $\langle 0, S_{\sigma_1} \rangle$ with S_{σ_1} referring to the set of jobs in σ_1. Then, only S_{σ_1} needs to be binary encoded into the hash value.

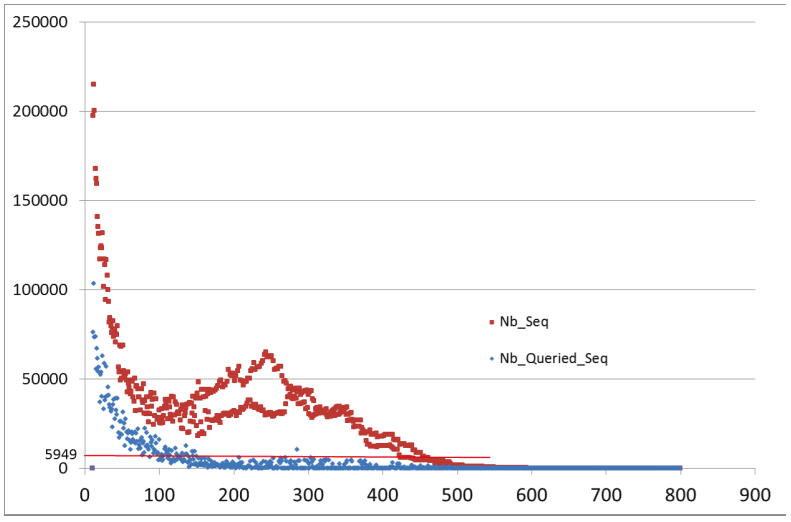
With respect to the database management, notice that when an element is added, in node memorization techniques, then the elements dominated by the added one are removed. Besides, due to memory limitation on the computer used for testing, we may need to clean the database when it is full on some instances. More precisely, in our experiments, the RAM is of 8Gb and hence the database size is also limited to 8Gb.

A cleaning strategy is needed to remove unpromising elements, i.e. those that are expected not to be used for pruning the search tree. As it is not clear which elements are unpromising, several strategies have been tested. We have implemented the following ones during our experimentations.

FIFO: First In First Out
This is one of the most common database cleaning strategy: when the memory is full, we first remove the first added elements. An extra structure is needed to record the order of elements according to the time when they are added. When the database if full and a long sequence is waiting to be inserted, it may be necessary to remove more than one elements in order to free enough space.

BEFO: Biggest Entry First Out
This cleaning strategy suggests to remove from the database the biggest elements (longest

sequences) in order to free enough continuous memory for storing new elements. For *solution memorization* it means removing nodes at higher levels in the search tree. An intuition of the impact of this cleaning strategy on *solution memorization* can be sketched from Figure 4.5 which presents the number of sequences memorized per size for an instance of the $1||\sum T_i$ scheduling problem with 800 jobs. It can be seen that sequences with "large number of jobs" (let's say more than 500 jobs) are not often used to prune nodes, and even if some large nodes could have been useful for node pruning, we may still expect that the solution of its subproblems generated by one or several branching can be found from the memory.



Nb_Seq: number of sequences of a given size, stored in the memory.
Nb_Queried_Seq: number of sequences of a given size, that are used to avoid solving twice identical problems.

Figure 4.5: Number of solutions and useful solutions in memory for an instance of $1||\sum T_i$ with 800 jobs

However, for *passive and predictive node memorization*, the strategy means removing nodes at lower levels of the search tree. These nodes refer to subproblems with many jobs already fixed (and memorized) and few jobs to schedule. It may be possible that the extra cost of memorizing a long fixed partial sequence is not inferior to solving the corresponding small subproblem directly without memorization. Since this is not obvious from a theoretic point of view, some preliminary experiments were performed in order to investigate whether it is better also to remove nodes at higher levels of the search tree in *node memorization*. Computational testing confirms that removing longest elements is always preferred, at least on problems $1|r_i|\sum C_i$, $1|\tilde{d}_i|\sum w_i C_i$ and $F2||\sum C_i$.

At each cleaning, we also tend to clean up a large amount of space in order to decrease the time cost induced by frequent cleaning operations.

LUFO: Least Used First Out

Figure 4.5 also suggests another cleaning strategy since a lot of sequences are never used

to prune nodes in the search tree. These sequences can be removed from the database to save space. To implement the LUFO cleaning strategy we keep an usage counter for each database element. The counter is incremented by 1 each time the element is queried and used to prune a node in the search tree, and it is decremented by 1 when a cleaning operation is performed. Elements whose counter is zero are removed by the cleaning operation. Note that in *node memorization*, when a database element is replaced by a new one, the latter should inherit the counter value of the old one. This is because that the counter value reflects the usefulness of a solution and the counter value of a newly added solution should not be smaller than the counter values of solutions that are dominated by the new one.

Preliminary results, not reported here, show that FIFO strategy is not efficient for the considered scheduling problems. BEFO strategy works better than FIFO, but its efficiency is not high enough to make a difference in the computational results. LUFO strategy is proved to be surprisingly efficient.

4.4 Application to the $1|r_i|\sum C_i$, $1|\tilde{d}_i|\sum w_i C_i$ and $F2||\sum C_i$ problems

In order to experiment the effectiveness of *Memorization* on scheduling problems, we first test it on three problems that were considered by T'kindt et al. (2004). In that work the authors used memory to apply the so-called *DP property* over nodes in order to prune the search tree. According to the memorization framework as defined in this chapter, what they have done is *passive node memorization* with a database cleaning strategy which replaces the shortest stored sequence by the new one when the database is full. The aim of that paper was also on choosing the most suitable search strategy when trying to solve these problems efficiently. In this section, for each of these three problems we apply the previously defined framework of *Memorization* with various considerations and discuss the obtained results. For each problem, we compare several *Branch & Bound* algorithms which are named according to their features: Depth-, Best- and Breadth- refer to *Branch & Bound* algorithms with the corresponding search strategies and no memorization included. Depth_X, Best_X and Breadth_X refer to a *Branch & Bound* algorithm with corresponding search strategies and *memorization* X used, with $X = S$ representing *solution memorization*, $X = Pa$ representing *passive node memorization* and $X = Pr$ the *predictive node memorization*. For *predictive node memorization*, we use *k-perm* heuristic to search for new sequences, as described in section 4.2.3.3.

k-perm heuristic also refers to a "dominance condition relying on scheduled jobs" as introduced by Jouglet et al. (2004). At a given node σS, assume that $\sigma = \sigma^0\sigma^k$ with σ^k the subsequence of the k last jobs in σ, k being an input parameter. The *k-perm* heuristic consists in enumerating all permutations of jobs in σ^k to obtain sequence σ^ℓ. Then, the first found sequence $\sigma^0\sigma^\ell$ dominating $\sigma^0\sigma^k$, if it exists, is used to prune node σS. The dominating sequence can be memorized. The notion of dominance between sequences is the one used to define the *check* function in *node memorization*. Preliminary tests suggest us to choose $k = 5$ in our implementations in order to have the most efficient *predictive node memorization scheme*.

Notice that *k-perm* search is not performed when *breadth first* strategy is used, since the memorization applied on active nodes already covers the effect of *k-perm*.

The algorithms proposed by T'kindt et al. (2004) in 2004 are also tested on the same dataset and they are named as Depth_Pa_04, Best_Pa_04 and Breadth_Pa_04, respectively. Compared to our algorithms Depth_Pa, Best_Pa and Breadth_Pa, the main differences are that the RAM usage is limited to 450M for Depth_Pa_04, Best_Pa_04 and Breadth_Pa_04, in order to obtain similar results to that reported in 2004. Also, in our algorithms LUFO is chosen as the database cleaning strategy.

The test results on instances of certain sizes are marked as *OOT* (out of time) if any of the instances is not solved after 5 hours. Analogously, with the application of *Memorization*, memory problems may occur and the limit on RAM usage may be reached, reported as *OOM* (out of memory). Note that according to our experiments, even when memory cleaning strategies are applied, *OOM* may still occur due to the fragmentation of the memory after a number of cleanings. Also note that LUFO is chosen as the cleaning strategy according to preliminary experimentations.

All tests have been done on a HP Z400 work station with 3.07GHz CPU and 8GB RAM. The datasets of all tests are open for public access from `http://www.vincent-tkindt.net/index.php/resources`.

4.4.1 Application to the $1|r_i|\sum C_i$ problem

The $1|r_i|\sum C_i$ problem asks to schedule n jobs on one machine to minimize the sum of completion times. Each job i has a processing time p_i and a release date r_i before which the job cannot be processed. The problem is NP-hard in the strong sense and it has been widely studied in the literature with both exact and heuristic algorithms considered. The referential computational results so far are done by T'kindt et al. (2004), in which with *forward branching* and *best first* and the application of a so-called *DP Property* the algorithm is able to solve instances with up to 130 jobs. A mixed integer programming approach is also reported by Kooli and Serairi (2014), which enables to solve instances with up to 140 jobs. However only 5 instances are generated in their experiments for each set of parameters. This makes their result less convincing due to the fact that the hardness of instances varies a lot even when generated with the same parameters, as observed during our study. Consequently, we consider in this section that the *Branch & Bound* algorithm provided by T'kindt et al. (2004) is at least as efficient as the approach used by Kooli and Serairi (2014). In contrast, 30 instances are generated by T'kindt et al. (2004) for each set of parameters , which leads to 300 instances for each size.

The work of T'kindt et al. (2004) uses the *Branch & Bound* algorithm of Chu (1992) as a basis, and so *forward branching* is adopted as the branching strategy. With respect to search strategies, *depth first*, *best first* and *breadth first* were all tested by T'kindt et al. (2004), aiming to explore the impact of different search strategies on the efficiency of the algorithm. The lower bounds and dominance conditions of Chu (1992) are kept. A so called *DP Property*, added as a new feature in the algorithm of T'kindt et al. (2004), is actually equivalent to *passive node memorization* in our terminology. The *check*() function is based on a dominance condition given by Chu (1992) and it was defined by T'kindt et al.

(2004) as follows:

$$check(\pi,\sigma) = \begin{cases} 1,\ if\ opt(\sigma|0) \leq opt(\pi|0)\ and \\ \quad opt(\sigma|0) + |\Omega| * E_{min}(\sigma) \leq opt(\pi|0) + |\Omega| * E_{min}(\pi) \\ 0,\ otherwise \end{cases} \tag{4.2}$$

with Ω the jobs that remain to be scheduled after sequence σ and π. We also have $E_{min}(\sigma) = \max(C(\sigma), \min_{r\in\Omega} r_i)$, with $C(\sigma)$ the completion time of σ. The item stored into memory is a tuple $\langle \sigma, C(\sigma), opt(\sigma|0), ExpAct\rangle$ and $E_{min}(\sigma)$ can be computed when needed. Note that this definition of *check* is an adaption of the general Equation 4.1 and if the *check* in Equation 4.1 return 1 then this *check* also returns 1.

4.4.1.1 Application of the memorization framework and improved results

The problem is not *decomposable* due to the existence of release dates. Therefore, with the choice of *forward branching*, *node memorization* should be chosen according to the decision tree in Figure 4.4. The lower bound used in the algorithm is based on the SRPT (Shortest Remaining Processing Time) rule. Together with the *check*() function defined in Equation 4.2, it is not clear whether the concordance property is answered. Hence, when *passive node memorization* is applied upon *best first*, all nodes need to be considered for the comparisons, while when it is applied with *breadth first*, only active nodes need to be considered. Therefore the choices made by T'kindt et al. (2004) with respect to memorization are kept. The *check*() function also remains the same, as defined in Equation 4.2.

Here we refresh the computational results of T'kindt et al. (2004) on new randomly generated input and also add results for *predictive node memorization* and *solution memorization*. The input is generated following the way described by Chu (1992), i.e. the processing times are generated uniformly from $[1, 100]$ and release dates are generated between 0 and $50.5 \cdot n \cdot r$ with r belonging to $\{0.2, 0.4, 0.6, 0.8, 1.0, 1.25, 1.50, 1.75, 2.0, 3.0\}$. 30 instances are generated for each value of r, hence leading to 300 instances for each size n from 70 to 140.

We first ran the algorithms of T'kindt et al. (2004) (Depth_Pa_04, Best_Pa_04 and Breadth_Pa_04) on these newly generated instances. All these three algorithms are able to solve instances with up to 110 jobs. On the running time, Best_Pa_04 is the fastest one, with a maximum running time of 27 seconds for instances with 110 jobs. This value becomes 40 seconds for Depth_Pa_04 and 1082 seconds for Breadth_Pa_04. It was reported by T'kindt et al. (2004) that the algorithms with best first and breadth first can solve respectively instances with 130 jobs and 120 jobs which is different from what we obtain. This reveals that the newly generated instances are harder than that of T'kindt et al. (2004), knowing that the computational power of our test environment is much better than in 2004. Moreover, we also observed that for Depth_Pa_04, the hardest instance of 110 jobs is solved in 40 seconds but the hardest instance of 90 jobs is solved in 1691 seconds. This is why we think that the hardness of instances vary a lot when generated randomly.

Results related to *node memorization* are put in Table 4.1. For all the three search strategies, *passive node memorization* enables to solve much larger instances with respect to the versions without memorization. This is sufficient to prove the power of memorization on this problem.

Depth_Pa, Best_Pa and Breadth_Pa are all more powerful than their counterparts of 2004, i.e. Depth_Pa_04, Best_Pa_04 and Breadth_Pa_04. This is especially visible on *depth first*, where Depth_Pa_04 solves instances with up to 110 jobs while Depth_Pa solves instances with up to 130 jobs. This makes Depth_Pa the global best algorithm and shows that when more physical memory is available and a larger database with an appropriate cleaning strategy is set, the memorization can be further boosted and the gain can be important.

The impact of *k-perm* search on this problem is very limited: *predictive node memorization* basically leads to the same result as *passive node memorization*. In addition, we also tested *solution memorization* on this problem since no theoretical dominance between the memorization schemes can be established for this problem. Since context dependent dominance conditions are enabled in the algorithm, we first disabled them in order to obtain the optimal solution of each node. But this turned out to be very inefficient. Therefore, we also implemented the memorization of lower bounds, as described in section 4.2.3.1. However, the resulting algorithm can only solve instances with up to 80 jobs, hence not competitive compared to *node memorization*, as predicted according to the decision tree in Figure 4.4.

It is also worth to be mentioned that the database cleaning strategy LUFO enables a faster solution of large instances. As an example, we found an instance with 140 jobs is solved in 1.6 hours by Depth_Pa with LUFO, while it needs 14 hours to be solved when the cleaning strategy of T'kindt et al. (2004) is kept instead. However, due to the hardness of another instance with 140 jobs, the algorithm Depth_Pa is finally out of time.

	n	70	80	90	100	110	120	130	140
Depth-	Navg	141247.8	1778751.2	OOT					
	Nmax	17491232	276190737						
	Tavg	1.8	22.4						
	Tmax	217	3238						
Depth_Pa	Navg	2583.4	5756.2	18639.9	26827.4	48502.9	174545.5	192409.4	OOT
	Nmax	147229	314707	2253897	644151	1281097	16575522	7742714	
	Tavg	0.0	0.0	0.3	0.7	1.3	7.1	9.1	
	Tmax	2	7	64	27	41	754	295	
Depth_Pr	Navg	1771.1	4455.1	12625.7	19621.7	30380.4	117865.6	128277.5	OOT
	Nmax	82765	267416	1455743	588429	1096520	11126694	5132228	
	Tavg	0.0	0.0	0.3	0.5	0.9	4.7	6.6	
	Tmax	1	7	46	28	39	488	252	
Best-		OOT							
Best_Pa	Navg	1230.5	3299.4	5235.1	9494.8	13658.5	38574.5	43986.9	OOT
	Nmax	36826	256534	292929	216293	228848	2675337	1449900	
	Tavg	0.0	0.2	0.2	0.4	0.6	15.3	11.8	
	Tmax	0	46	38	27	25	3595	1630	
Best_Pr	Navg	1229.6	3298.2	5229.0	9490.7	13545.7	38560.1	43989.8	OOT
	Nmax	36826	256529	292927	216037	228832	2674776	1449872	
	Tavg	0.0	0.2	0.2	0.4	0.7	15.4	11.9	
	Tmax	1	47	39	28	25	3579	1636	
Breadth-		OOT							
Breadth_Pa	Navg	1947.7	6745.0	9893.8	21308.5	27383.1	OOT		
	Nmax	90494	709607	733980	575430	1209481			
	Tavg	0.0	4.6	3.4	5.3	5.7			
	Tmax	9	1319	897	483	935			

Table 4.1: Results of new algorithms on the $1|r_i|\sum C_i$ problem

4.4.2 Application to the $1|\tilde{d}_i|\sum w_i C_i$ problem

The $1|\tilde{d}_i|\sum w_i C_i$ problem asks to schedule n jobs on a single machine. Each job i has a processing time p_i, a weight w_i and a deadline $\tilde{d}_i$ which has to be answered. The objective is to minimize the total weighted completion time $\sum w_i C_i$. The problem is NP-hard in the strong sense and has been solved by *Branch & Bound* algorithms (Posner, 1985; Potts and Van Wassenhove, 1983), with the algorithm of Posner (1985) being slightly superior. The basic algorithm described by T'kindt et al. (2004) is a combination of algorithms of Posner (1985); Potts and Van Wassenhove (1983) by incorporating the lower bound and the dominance condition of Posner (1985) into the *Branch & Bound* algorithm of Potts and Van Wassenhove (1983). With respect to search strategies, all the three strategies, i.e. *depth first*, *best first* and *breadth first* were considered by T'kindt et al. (2004) and *backward branching* is adopted as the branching scheme as in the algorithms of Posner (1985); Potts and Van Wassenhove (1983). Similarly to what is done on the $1|r_i|\sum C_i$ problem, the *DP Property* is also considered by T'kindt et al. (2004), which is actually *passive node memorization*. The *check*() function is defined as follows, where Ω is the set of jobs to be scheduled before σ and π.

$$check(\pi,\sigma) = \begin{cases} 1, \ if \ opt(\sigma|\sum_{i\in\Omega} p_i) \leq opt(\pi|\sum_{i\in\Omega} p_i) \\ 0, \ otherwise \end{cases} \tag{4.3}$$

The items stored in the database are $\langle \sigma, opt(\sigma|\sum_{i\in\Omega} p_i), ExpAct\rangle$. In 2003, Pan (2003) proposed another *Branch & Bound* algorithm with reported experiments showing that it can solve to optimality all instances with up to 90 jobs in size. As the testing protocol is identical to the one used in 2004 by T'kindt et al. (2004), we can conclude that the algorithm of Pan is outperformed by the best one proposed by T'kindt et al. (2004) (which is reported as being able to solve instances with up to 130 jobs in size).

4.4.2.1 Application of the memorization framework and improved results

This problem is *decomposable* according to Definition 7. From the decision tree in Figure 4.4 we can derive that with the depth first search strategy, *solution memorization* should be considered, even though its superiority over *node memorization* depends on the presence of *context dependent dominance conditions* in the algorithm. In the paper of T'kindt et al. (2004), *node memorization* was implemented with that strategy. Consequently, in this section we compare four *Branch & Bound* algorithms: the three versions of T'kindt et al. (2004), i.e. *node memorization* applied to the three search strategies and a version based on depth first with *solution memorization*.

The concordance property is answered (see Proposition 11) and hence the *passive node memorization* only considers explored node when the search strategy is *best first*, and only active nodes need to be considered in *breadth first*. For *solution memorization*, the items stored into the memory are $\langle \pi, opt(\pi|0)\rangle$. For *node memorization*, the *check*() function and the items stored remain the same as in the work of T'kindt et al. (2004), as described in the previous section.

About *solution memorization*, context dependent dominance conditions are enabled in the algorithm. Their removal has been experimentally proved to lead to an inefficient algorithm. Therefore, lower bounds are memorized during the *solution memorization*, as described in section 4.2.3.1.

Proposition 11. *With the check() function defined in Equation 4.3, our algorithms verify the concordance property (Property 6).*

Proof. Consider two nodes $S\sigma$ and $S\pi$. First notice that the subproblem to solve in both nodes are the same, which consists in scheduling jobs from S starting from time 0. The lower bound used in the algorithm (see Posner (1985); Potts and Van Wassenhove (1983)) returned on the subproblems on S are the same for the two nodes. Therefore, if $check(\pi, \sigma) = 1$, which means $opt(\sigma|\sum_{i\in\Omega} p_i) \leq opt(\pi|\sum_{i\in\Omega} p_i)$, then $LB(S\sigma) \leq LB(S\pi)$.

With the same reasoning, if $LB(S\sigma) \leq LB(S\pi)$, it can be deduced that the relation $opt(\sigma|\sum_{i\in\Omega} p_i) \leq opt(\pi|\sum_{i\in\Omega} p_i)$ must hold, and hence $check(\pi, \sigma) = 1$. □

Following the test plan described by Potts and Van Wassenhove (1983), for each job i, its processing time p_i is an integer generated randomly from the uniform distribution $[1, 100]$ and its weight w_i is generated uniformly from $[1, 10]$. The total processing time $P = \sum_{i=1}^{n} p_i$ is then computed and for each job i an integer deadline d_i is generated from the uniform distribution $[P(L - R/2), P(L + R/2)]$, with L increase from 0.6 to 1.0 in steps of 0.1 and R increases from 0.2 to 1.6 in steps of 0.2. In order to avoid generating

infeasible instances, a (L, R) pair is only used when $L+R/2 > 1$, hence only 20 (L, R) pairs are actually used, for each of which 10 feasible instances are generated, yielding a total of 200 instances for each value of n from 40 to 140. We first present the results of *passive node memorization* algorithms (Depth_Pa_04, Best_Pa_04 and Breadth_Pa_04) from T'kindt et al. (2004). Both Depth_Pa_04 and Best_Pa_04 are stated by T'kindt et al. (2004) to solve instances with up to 110 jobs. However, they are only capable of solving instances with 70 jobs on the newly generated instances, with a maximum solution time 11 seconds and 285 seconds, respectively. Breadth_Pa_04 was reported to be able to solve instances with up to 130 jobs in 2004 but this falls down to 100 jobs in our tests with a maximum solution time of 36 seconds. This difference is not negligible and it reveals the fact that the newly generated instances seem much harder than those generated by T'kindt et al. (2004).

The results of the new algorithms are presented in Table 4.2. On *depth first*, without *memorization* the program is "out of time" on instances with 50 jobs, while both *solution memorization* and *passive node memorization* enable to solve instances with up to 100 jobs, with *passive node memorization* running faster. With the activation of *k-perm* search, Depth_Pr enables to solve 30 more jobs than Depth_Pa. This strongly proves the power of all the three memorization schemes. It also worth to be noticed that Depth_Pa solves instances with 30 more jobs with respect to Depth_Pa_04, knowing that the only differences between these two algorithm are that the database size in Depth_Pa is larger and the database cleaning strategy is different.

For *best first*, the same phenomenon can be observed, that is, Best_Pr is more efficient than Best_Pa, which is better than Best- and Best_Pa_04. Best_Pr can also solve instances with up to 130, and faster than Depth_Pr.

Breadth_Pa is the most powerful algorithm among all. It is surprising to see that without *memorization* Breadth- cannot even solve all instances of 40 jobs, while with *passive node memorization* instances of 130 jobs are all solved in an average solution time of 27 seconds. Again, as for the $1|r_i|\sum C_i$ problem, LUFO allows to accelerate the solution but it did not enable to solve larger instances.

	n	40	50	60	70	80	90	100	110	120	130	140
Depth-	Navg	116827.7	OOT									
	Nmax	14536979										
	Tavg	1										
	Tmax	74										
Depth_S	Navg	772.0	2718.0	6706.0	28463.0	114970.0	139382.0	563209.0	OOT			
	Nmax	17699	60462	137207	1660593	6180097	2803714	12335703				
	Tavg	0.4	0.5.0	1.0	3.0	15.0	12.0	113.0				
	Tmax	1	2	6	275	1544	474	5346				
Depth_Pa	Navg	559.1	2091.3	5240.1	20068.3	75727.4	139429.8	376206.9	OOT			
	Nmax	11963	83075	94189	1004546	1960891	4321070	5549747				
	Tavg	0.4	0.4	0.5	0.9	2.9	5.5	17.0				
	Tmax	1	1	2	39	157	312	515				
Depth_Pr	Navg	326.4	901.7	2184.1	6825.3	20429.0	32531.0	90375.3	266689.8	574824.4	1397463.6	OOT
	Nmax	3431	17447	28677	187425	665376	768802	1781123	14713483	11236833	103699138	
	Tavg	0.4	0.4	0.4	0.6	1.0	1.0	3.8	14.4	37.9	108.2	
	Tmax	0	1	1	5	30	21	51	901	1255	8732	
Best-		OOT										
Best_Pa	Navg	334.6	879.3	1859.0	7159.9	16581.8	27259.5	60349.0	OOT			
	Nmax	3800	20889	25574	440623	547165	1252600	798372				
	Tavg	0.4	0.4	0.4	0.7	1.3	1.8	4.3				
	Tmax	0	1	1	30	73	130	91				
Best_Pr	Navg	292.0	708.9	1486.9	4312.7	10301.7	14642.9	31891.2	145203.1	239837.4	330474.1	OOT
	Nmax	2435	11762	18051	120259	319068	276507	332022	10659343	7578570	6712266	
	Tavg	0.4	0.4	0.4	0.5	0.8	1.0	1.9	45.6	20.0	26.7	
	Tmax	0	1	1	4	23	13	25	5008	1137	716	
Breadth-		OOM										
Breadth_Pa	Navg	348.6	940.9	1833.1	6533.4	14964.4	23725.0	53309.2	102633.5	239512.5	329902.3	OOT
	Nmax	4701	16952	24559	437697	453506	868876	789310	5975094	7577492	6702080	
	Tavg	0.0	0.0	0.0	0.2	0.4	0.8	2.0	10.1	20.0	26.7	
	Tmax	0	0	0	9	15	31	36	1353	1135	718	

Table 4.2: Results of the new algoritihms on the $1|\tilde{d}_i|\sum w_i C_i$ problem

4.4.3 Application to the $F2||\sum C_i$ problem

The $F2||\sum C_i$ problem asks to schedule n jobs are to be scheduled on two machines M_1 and M_2. Each job i needs first to be processed on M_1 for $p_{1,i}$ time units then be processed on M_2 for $p_{2,i}$ time. The objective is to minimize the sum of completion times of jobs. We restrict to the set of permutation schedules in which there always exist an optimal solution. A permutation schedule is a schedule in which the jobs sequences on the two machines are the same. The problem is NP-hard in the strong sense. Up to 2016, the best exact algorithm was the *Branch & Bound* algorithm proposed by T'kindt et al. (2004) and based on the *Branch & Bound* algorithm of Della Croce et al. (2002). Recently, Detienne et al. (2016) proposed a new and very efficient *Branch & Bound* algorithm capable of solving instances with up to 100 jobs in size. This is definitely the state-of-the-art exact method for solving the $F2||\sum C_i$ problem. However, in order to evaluate the impact of using *Memorization* in a *Branch & Bound* algorithm we make use of the algorithms described by T'kindt et al. (2004) since their code was directly available to us.

The adopted branching scheme in this algorithm is *forward branching* and all the three search strategies were considered. The *DP Property* is also considered by T'kindt et al. (2004), which is actually *passive node memorization*. The *check*() function is based on a result reported by Della Croce et al. (2002) and is defined as follows:

$$check(\pi,\sigma) = \begin{cases} 1, \ if \ opt(\sigma|0) \leq opt(\pi|0) \ and \ |\Omega| * (C_2(\sigma) - C_2(\pi)) \leq opt(\pi) - opt(\sigma) \\ 0, \ otherwise \end{cases} \tag{4.4}$$

where Ω is the set of jobs to be scheduled after σ and π, $C_2(\cdot)$ is the completion time of a given sequence on the second machine. The items stored into the database are $\langle \sigma, C_2(\sigma), opt(\sigma|0), ExpAct \rangle$.

4.4.3.1 Application of the memorization framework and improved results

This problem is not *decomposable* since given a partial solution of the form σS with σ a fixed sequence, the optimal solution of subproblem S depends on the order of jobs in σ. From the decision tree in Figure 4.4 we can derive that with the depth first search strategy, *solution memorization* should be considered, even though its superiority over *node memorization* depends on the presence of *context dependent dominance conditions*. *Node memorization* was implemented with that strategy by T'kindt et al. (2004). Consequently, in this section we compare four *Branch & Bound* algorithms: the three versions of T'kindt et al. (2004), i.e. *node memorization* applied to the three search strategies and a version based on depth first with *solution memorization*.

With the *check*() function defined in Equation 4.4 and the lower bound (a Lagrangian Relaxation based lower bound) used in the algorithm, the concordance property is not answered. We performed experiments to look for the case where for two nodes σS and πS, $check(\pi,\sigma) = 1$ but $LB(\pi) < LB(\sigma)$ and we found it. Therefore, the concordance property is not verified and both active and explored nodes need to be considered for *best first* strategy. For *breadth first* strategy, only active nodes need to be considered.

For *solution memorization*, since context dependent dominance conditions are enabled in the algorithm, and they are important for a fast solution of the problem, lower bounds are memorized during the *solution memorization*, as described in section 4.2.3.1. The items stored into the memory are $\langle \pi, t_1, t_2, C_2(\pi), opt(\pi|(t_1,t_2)) \rangle$ where t_1 is the actual starting time of π on the first machine and t_2 is the actual starting time of π on the second machine. Besides, $opt(\pi|(t_1,t_2))$ is the sum of completion times of jobs in π, when π starts at time t_1 on the first machine and time t_2 on the second machine. For *node memorization*, the *check*() function and the stored item remain the same as in the work of T'kindt et al. (2004), as described in the previous section.

30 instances are generated for each size n from 10 to 40, with the processing times generated randomly from an uniform distribution in $[1, 100]$. Again, we ran the algorithms of T'kindt et al. (2004) on these newly generated instances. Depth_Pa_04 is able to solve instances of 40 jobs which is 5 jobs more than reported by T'kindt et al. (2004), with a maximum solution time about 3.4 hours. Both Best_Pa_04 and Breadth_Pa_04 solve instances with up to 35 jobs, as reported by T'kindt et al. (2004), with maximum solution times of 43 seconds and 806 seconds, respectively.

Other results are given in Table 4.3. Depth- is able to solve instances with 35 jobs. Best- is able to solve instances with 30 jobs and Breadth- can only solve up to 25 jobs. With *passive node memorization* enabled, Depth_Pa solves instances with 5 more jobs than

Depth-. Best_Pa and Breadth_Pa solve instances with 10 more jobs than the versions without *Memorization*. With respect to algorithms X_Pa_04, algorithms X_Pa use a larger database, more precisely, the maximum number of solutions that can be stored is set to 6000000 instead of 350000. This enables Best_Pa to solve 5 more jobs than Best_Pa_04. However, Depth_Pa and Breadth_Pa are not able to solve larger instances with respect to Depth_Pa_04 and Breadth_Pa_04, even though they solve instances faster. Notice that the $F2||\sum C_i$ problem is a really hard problem, certainly more difficult than the two other problems previously tackled.

Also, the LUFO strategy is adopted for database cleaning but it did not enable to solve larger instances without having an "Out of Time" problem.

Predictive node memorization is not more efficient than *passive node memorization*: in fact no nodes are cut by undertaking a *k-perm* search. The result is hence even slightly slower due to the time consumed by the call to the *k-perm* heuristic. Depth_S solve instances with less nodes generated compared to Depth-. However, its efficiency is even less than Depth-, due to the processing of lower bound memorization.

From a global point of view, the power of memorization is also illustrated on this problem, since we always have benefits in using it. As a perspective for this problem it could be interesting to evaluate the contribution of *Memorization* when embedded into the state-of-the-art algorithm of Detienne et al. (2016).

	n	10	15	20	25	30	35	40	45
Depth-	Navg	23.7	255.6	4137.7	21460.4	317102.0	3615780.0	OOT	
	Nmax	84	2367	83863	311742	3097479	53187978		
	Tavg	0.0	0.0	0.1	0.8	26.0	423.0		
	Tmax	0	0	2	17	248	6128		
Depth_S	Navg	24.0	228.0	3561.0	19733.0	294355.0	3425633.0	OOT	
	Nmax	84	1735	68070	273146	2712580	49360565		
	Tavg	0.0	0.0	0.1	1.0	29.0	497.0		
	Tmax	0	0	2	15	248	6933		
Depth_Pa	Navg	22.8	187.2	1573.0	8205.0	61337.0	337194.0	1894037.2	OOT
	Nmax	80	1083	17114	48459	291750	1568506	15472612	
	Tavg	0.0	0.0	0.0	0.1	4.1	35.0	328.3	
	Tmax	0	0	0	2	21	163	3627	
Depth_Pr	Navg	22.8	187.2	1573.0	8205.0	61361.3	337194.0	1894037.0	OOT
	Nmax	80	1083	17114	48459	291016	1568506	15472612	
	Tavg	0.0	0.0	0.0	0.1	4.1	32.8	332.8	
	Tmax	0	0	0	2	23	173	3664	
Best-	Navg	23.7	249.3	3993.1	21717.7	291131.9	OOM		
	Nmax	84	2253	83863	311742	2451152			
	Tavg	0.0	0.0	0.1	0.7	19.1			
	Tmax	0	0	2	17	197			
Best_Pa	Navg	20.9	139.5	957.3	4780.7	28957.0	112229.8	495186.5	OOM
	Nmax	72	624	6646	21022	152797	426641	3617824	
	Tavg	0.0	0.0	0.0	0.0	1.2	7.8	80.6	
	Tmax	0	0	0	1	4	43	1253	
Best_Pr	Navg	20.9	139.5	957.3	4780.7	28957.0	112229.8	495186.5	OOM
	Nmax	72	624	6646	21022	152797	426641	3617824	
	Tavg	0.0	0.0	0.0	0.0	1.4	8.3	83.1	
	Tmax	0	0	0	1	5	45	1283	
Breadth-	Navg	23.9	266.1	5181.8	39303.6	OOT			
	Nmax	84	2360	83863	311742				
	Tavg	0.0	0.0	0.1	1.6				
	Tmax	0	0	2	17				
Breadth_Pa	Navg	21.0	148.8	1369.5	8889.1	115219.2	345109.6	OOT	
	Nmax	72	692	9927	63485	2242263	2357023		
	Tavg	0.0	0.0	0.0	0.2	26.1	54.2		
	Tmax	0	0	0	3	711	665		

Table 4.3: Results of new algorithms on the $F2||\sum C_i$ problem

4.5 Application to the $1||\sum T_i$ problem

In this section, we report the results of the application of *Memorization* on solving the $1||\sum T_i$ problem, already tackled in chapter 3 by *Branch & Merge* on the theoretical aspect. We first recall some main properties of the problem, then determine parameters for *Memorization* and finally report the computational results.

4.5.1 Preliminaries

The $1||\sum T_i$ problem asks to schedule a set of n jobs $N = \{1, 2, \ldots, n\}$ on a single machine to minimize the sum of tardiness. The current state-of-the-art exact method in practice is a *Branch & Bound* algorithm (named as BB2001 in this chapter) which solves to optimality problems with up to 500 jobs in size (Szwarc et al., 2001). The main properties of the problem have already been introduced in section 3.1. Some of them are reminded below for the ease of reference.

Let $(1, 2, \ldots, n)$ be a LPT (Longest Processing Time first) sequence and $([1], [2], \ldots, [n])$ be an EDD (Earliest Due Date first) sequence of all jobs.

We first introduce two important decomposition properties, which are the same as Property 1 and Property 2, respectively, from chapter 3.

Decomposition 1. *(Lawler, 1977) (Lawler's decomposition) Let job* 1 *in LPT sequence correspond to job* $[k]$ *in EDD sequence. Then, job* 1 *can be set only in positions* $h \geq k$ *and the jobs preceding and following job* 1 *are uniquely determined as* $B_1(h) = \{[1], [2], \ldots, [k-1], [k+1], \ldots, [h]\}$ *and* $A_1(h) = \{[h+1], \ldots, [n]\}$.

Decomposition 2. *(Szwarc et al., 1999) Let job* k *in LPT sequence correspond to job* [1] *in EDD sequence. Then, job* k *can be set only in positions* $h \leq (n-k+1)$ *and the jobs preceding job* k *are uniquely determined as* $B_k(h)$*, where* $B_k(h) \subseteq \{k+1, k+2, \ldots, n\}$ *and* $\forall i \in B_k(h), j \in \{n, n-1, \ldots, k+1\} \setminus B_k(h)$*,* $d_i \leq d_j$

The two above decomposition rules can be applied simultaneously to derive a decomposing branching scheme called *Double Decomposition* (Szwarc et al., 2001). At any node, let S_i be a set of jobs to schedule. Note that some other jobs may have already been fixed on positions before or after S_i, implying a structure like $\sigma_1 S_1 \sigma_2 S_2 ... \sigma_i S_i ... \sigma_k S_k$ over all positions, but a node only focuses on the solution of one subproblem, induced by one subset of jobs (S_i here). With depth first, which is the search strategy retained in the *Branch & Bound* BB2001, the *Double Decomposition* is always applied on S_1. This works as follows. First find the longest job ℓ and the earliest due date job e in S_1. Then apply Decomposition 1 (resp. Decomposition 2) to get the lists L_ℓ (resp. L_e) of positions, on which ℓ (resp. e) can be branched on. As an example, suppose $L_e = \{1, 2\}$ and $L_\ell = \{5, 6\}$. Then, a double branching can be done by fixing job e on position 1 and fixing job ℓ on position 5, decomposing the jobset S_i to three subsets (subproblems): the jobs before jobs job e, which is $\emptyset$; the jobs between e and ℓ; and finally the jobs after ℓ. In the same way, the other 3 branching can be performed by fixing jobs e and ℓ in all compatible position pairs: $(1, 6)$, $(2, 5)$ and $(2, 6)$.

When branching from a node, another particular decomposition may occur as described in Property 12. Assume that a given subset of jobs S is decomposed into two disjoint subsets B and A, with $B \cup A = S$ and all jobs of B are scheduled before that of A in an optimal schedule of S: (B, A) is then called an optimal block sequence and Property 12 states when does such decomposition occur. In that case Decomposition 1 and Decomposition 2 are not applied but two child nodes are rather created each one corresponding to one block of jobs (A or B), following Property 12 (also called the *split* property).

Let E_j and L_j be the earliest and latest completion times of job j, that is if B_j (resp.

A_j) is the currently known jobset that precedes (resp. follow) job j, then $E_j = p(B_j) + p_j$, and $L_j = p(N \setminus A_j)$.

Property 12. *(Szwarc et al., 1999) (Split)*
(B, A) is an optimal block sequence if $\max_{i \in B} L_i \leq \min_{j \in A} E_j$.

The value of E_i and L_i of each job i can be obtained by applying Emmons' conditions (Emmons, 1969) following the $\mathcal{O}(n^2)$ procedure provided by Szwarc et al. (1999).

An initial version of *solution memorization* has been already implemented in BB2001, even though it was called *Intelligent Backtracking* by the authors. Remarkably, lower bounds are not used in this *Branch & Bound* algorithms due to the "Algorithmic Paradox" (Paradox 1) found by Szwarc et al. (2001). This one shows that the power of *Memorization* largely surpasses the power of the lower bounding procedures in the algorithm.

Paradox 1. *"...deleting a lower bound drastically improves the performance of the algorithm..."*

Paradox 1 is simply due to the fact that a lot of identical subproblems occur during the exploration of the search tree. The computation time required by lower bounding procedures to cut these identical problems are much higher than simply solving that subproblems once, memorizing the solution and reusing it whenever the subproblem appears again. Besides, pruning nodes by the lower bound may negatively affect memorization since the nodes that are cut cannot be memorized.

The BB2001 algorithm uses a depth first strategy and for each node to branch on, the following procedure is applied:

1. Search the solution of the current problem, defined by a set of jobs and a starting time of the schedule, in "memory", and return the solution if found; otherwise go to 2.

2. Use Property 12 to split the problem and solve each new subproblem recursively starting from step 1. If no split can be done, go to step 3.

3. Combine Decompositions 1 and 2 to branch on the longest job and the smallest-due-date job to every candidate positions. For each branching case, solve subproblems recursively, then store in memory the best solution among all branching cases and return it.

The complexity of this algorithm has been analyzed in section 3.5.1 as an additional result. Note that due to Paradox 1, all lower bounding procedures are removed, which makes the *Branch & Bound* algorithm a simple branching algorithm. Notice that *solution memorization* can be implemented in BB2001 as suggested in section 4.3. In BB2001, when the database of stored solutions is full, no cleaning strategy is used and no more partial solutions can be stored. The memory limit of this database in BB2001 is not mentioned by Szwarc et al. (2001).

4.5.2 Application of the memorization framework and improved results

We take the reference algorithm BB2001 as a basis, in which *decomposition branching* and *solution memorization* are already chosen. The *decomposition branching* has been proved to be very powerful, and there is no evidence that other branching schemes like *forward branching* or *backward branching* can lead to a better algorithm (see Szwarc et al. (2001)). The problem is *decomposable* according to Definition 7. The main discussion relies on the relevancy of considering *node memorization* instead of *solution memorization*. As already mentioned in section 4.2.4.4, it is not obvious to implement *node memorization*, for a decomposing branching scheme, which could outperform the *solution memorization*. Here a node is structured as $\sigma_1 S_1 ... \sigma_k S_k$ with the $\sigma_i's$ being the partial sequences to memorize in *node memorization*. Assume we have two nodes $\sigma_1 S_1 ... \sigma_k S_k$ and $\pi_1 S'_1 ... \pi_\ell S'_\ell$, it is not apparent to find σ_i and π_j, $i \in \{1, .., k\}$, $j \in \{1, .., \ell\}$, such that σ_i and π_j are of same jobs and have the same starting time. Moreover it seems complicated to design an efficient *check*() function deciding which of these two nodes is dominating the other. We found no way to implement *node memorization* which could hopefully lead to better results than those obtained with *solution memorization*. Consequently, *solution memorization* only is considered and, as sketched in sections 4.2.4.5 and 4.2.4.6, there is no interest in considering *best first* or *breadth first* search strategies.

Henceforth, the choices done by Szwarc et al. (2001) with respect to memorization were good choices. In the remainder we investigate limitations of the memorization technique as implemented by Szwarc et al. (2001) and propose improvements which significantly augment the efficiency of the algorithm.

Our algorithm is based on BB2001, with two main changes.

Since the memory usage was declared as a bottleneck of BB2001, we firstly retest BB2001 on our machine: a HP Z400 work station with 3.07GHz CPU and 8GB RAM. 200 instances are generated randomly for each problem size using the same generation scheme as in the paper of Potts and Van Wassenhove (1982). Processing times are integers generated from an uniform distribution in the range $[1, 100]$ and due dates d_i are integers from a uniform distribution in the range $[p_i u, p_i v]$ where $u = 1-T-R/2$ and $v = 1-T+R/2$. Each due date is set to zero whenever its generated value is negative. Twenty combinations (R, T) are considered where $R \in \{0.2, 0.4, 0.6, 0.8, 1\}$, and $T \in \{0.2, 0.4, 0.6, 0.8\}$. Ten instances are generated for each combination and the combination $(R = 0.2, T = 0.6)$ yields the hardest instances as reported in the literature (see Szwarc et al. (1999)) and confirmed by our experiments. Again, all data can be found from `http://www.vincent-tkindt.net/index.php/resources`.

Table 4.4 presents the results we obtain when comparing different algorithms. For each version we compute the average and maximum CPU time T_{avg} and T_{max} in seconds for each problem size. The average and maximum number of explored nodes N_{avg} and N_{max} are also computed. The time limit for the solution of each instance is set to 4 hours, and the program is considered as OOT (Out of Time) if it reaches the time limit. Also, when memorization is enabled without a database cleaning strategy, the physical memory may be saturated by the program, in which case the program is indicated as OOM (Out of Memory).

Our implementation of BB2001 solves instances with up to 900 jobs in size as reported in Table 4.4, with an average solution time of 764s and a maximum solution time of 9403s for 900-job instances, knowing that the original program, as tested in 2001 was limited to instances with up to 500 jobs due to memory size limit. Their tests were done on a Sun Ultra-Enterprise Station with a reduced CPU frequency (<450MHz) and a RAM size not stated. It is anyway interesting to see that with just the computer hardware evolution, *Memorization* is augmented to solve instances with 400 jobs more.

BB2001 is out of time (>4h) for instances with 1000 jobs, and the memory size seems no longer to be the bottleneck. The first improvement we propose presume on the vein of Paradox 1.

Paradox 2. *Removing Split procedure (Property 12) from BB2001 drastically accelerate the solution.*

The effect of Paradox 2 is astonishing. The resulting algorithm *NoSplit* solves instances with 700 jobs with an average solution time 20 times faster: from 192 seconds to 9 seconds (see Table 4.4). In fact, *Split* is performed based on precedence relations between jobs, induced by the computation of the $E_j's$ and $L_j's$. The computation of these precedence relations is time consuming in practice. Moreover, as already claimed, many identical problems appear in the search tree and the *Split* procedure in BB2001 is run each time. When *Split* is removed, identical problems are solved needing more time when first met, but then never solved twice thanks to *solution memorization*. However, the disadvantage is also clear: more solutions are added to the database and hence the database is filled faster than when *Split* is enabled. This is why *NoSplit* encounters memory problems on instances with 800 jobs. This was not considered by Szwarc et al. (2001) because *Split* is a very strong component of the algorithm and the computer memory at that time also discourages this tentative. S_i, SDD_2

At this point, we have a better understanding of the power of *solution memorization* on this problem and we become curious on the effectiveness of memorized solutions. In other words, what are the stored solutions that are effectively used? To answer this question, we test cleaning strategies as defined in section 4.3, to remove useless solutions when the database memory is "full". The most efficient strategy is proved to be LUFO by preliminary experiments not reported here. Embedding such a memory cleaning strategy is our second contribution to BB2001 algorithm.

In Table 4.4, the final implementation of the memorization mechanism within the *Branch & Bound* algorithm for the $1||\sum T_i$ problem is referred to as NoSplit_LUFO. All 200 instances with 1200 jobs are solved, with an average solution time of 192 seconds, while BB2001 is limited to instances with 900 jobs.

The experiments presented so far have shown that correctly tuning the memorization mechanism, notably by considering a cleaning strategy and studying interference with other components of the algorithm may lead to serious changes of its efficiency. However, the striking point of these experiments relates on the comparison between the version of BB2001 without the memorization mechanism (algorithm Depth-) and NoSplit_LUFO. Table 4.4 highlights the major contribution of memorization: Depth- being limited to instances with

	n	300	400	500	600	700	800	900	1000	1100	1200	1300
Depth-	Navg	46046201	OOT									
	Nmax	2249342615										
	Tavg	155										
	Tmax	6499										
BB2001	Navg	61501	136452	290205	560389	880268	1534960	2053522	OOT			
	Nmax	663268	1884993	3585456	5784871	9802077	18199764	19352429				
	Tavg	2	9	31	85	192	469	763				
	Tmax	33	193	580	1263	2963	6817	9403				
NoSplit	Navg	202970	457918	985235	1934818	3053648	OOM					
	Nmax	2156144	6027604	13028651	20285112	33977553						
	Tavg	0	0	2	4	9						
	Tmax	4	13	34	59	114						
NoSplit_LUFO	Navg	202970	457918	985235	1934818	3086620	5408511	7697810	12578211	19100285	28223766	OOT
	Nmax	2156144	6027604	13028651	20285112	36853477	60151076	88909109	139698961	332937242	420974965	
	Tavg	0	0	2	5	9	20	31	61	112	192	
	Tmax	4	13	34	61	136	275	429	832	2504	3763	

Table 4.4: Results for the $1||\sum T_i$ problem

up to 300 jobs while NoSplit_LUFO is capable of solving all instances with 1200 jobs. It is evident that memorization is a very powerful mechanism.

4.6 Chapter summary

In this chapter we focus on the application of *Memorization* within branching algorithms for the efficient solution of sequencing problems. A framework of *Memorization* is established with several memorization schemes defined. Advices are provided to choose the best memorization approach according to the branching scheme and the search strategy of the algorithm. Some details on the efficient implementation of *Memorization* are also discussed.

The application of the framework has been done on four scheduling problems. Even if the impact of *Memorization* depends on the problem, for all the tackled problems it was beneficial to use it. Table 4.5 provides a summary of the conclusions obtained.

Problem	Largest instances solved		Features of the best algorithm with memorization	Best in literature?
	Without memorization	With memorization		
$1\|r_i\|\sum C_i$	80 jobs	130 jobs	depth first+ *predictive node memorization*	yes
$1\|\tilde{d}_i\|\sum w_iC_i$	40 jobs	130 jobs	breadth first+ *passive node memorization*	yes
$F2\|\|\sum C_i$	30 jobs	40 jobs	best first+ *passive node memorization*	no
$1\|\|\sum T_i$	300 jobs	1200 jobs	depth first+ *solution memorization*	yes

Table 4.5: Conclusions on the tested problems

Fundamentally, what we call the *Memorization Paradigm* relies on a simple but potentially very efficient idea: avoid solving multiple times the same subproblems by memorizing what has already been done or what can be done in the rest of the solution process. The contribution of this paradigm strongly relies on the branching scheme which may induce more or less redundancy in the exploration of the solution space. Noteworthy, the four scheduling problems dealt with in this chapter mainly serve as applications illustrating how memorization can be done in an efficient way. But, it is also clear that it can be applied to other hard combinatorial optimization problems, by the way making this contribution interesting beyond scheduling theory. To our opinion, the memorization paradigm should be embedded into any branching algorithm, so creating a new class of branching algorithms called *Branch & Memorize* algorithms. They may have a theoretical interest by offering the possibility of reducing the worst-case time complexity with respect to *Branch & Bound* algorithms. And they also have an interest from an experimental viewpoint, as illustrated in this chapter.

The main work in this chapter has been performed together with Vincent T'Kindt and Federico Della Croce (*Politechnico di Torino*, Italy). The computational results on the $1||\sum T_i$ problem have been reported at the international conference on *Industrial Engineering and Systems Management* (IESM 2017) (Shang et al., 2017e), being selected as the first prize of the *Best Student Paper Award*. It has also been communicated in the

ROADEF 2017 conference (Shang et al., 2017b) and the joint working days (26th/27th, September 2017) of the french GOThA and Bermudes groups. The complete results have been submitted to the *European Journal of Operational Research* and it can be found on HAL (Shang et al., 2017d).

Conclusion

Three main contributions are worked out during this thesis. The first one, presented in chapter 2, is about a *Dynamic Programming* algorithm which solves the $F3||C_{max}$ problem in $\mathcal{O}^*(3^n)$ time and space. The algorithm is based on a local dominance condition, consisting of two inequalities of the completion time on the second and the third machine. For any two partial solution sequences satisfying the condition, the dominated one is eliminated, therefore no longer considered during the later construction of the optimal global solution. The algorithm can be seen as a generalization of the conventional DP solving TSP-like problems, with the difference that in conventional DP the partial solutions of an optimal solution is optimal with respect to their corresponding subproblems, while this is not the case for flowshop problems. The practical efficiency of the algorithm has been evaluated. Without much surprise, the algorithm is proved to be impractical due to its exponential space complexity. This is a common drawback of *Dynamic Programming* algorithms over all subsets. Its running time on randomly generated instances is also far from competitive compared to state-of-the-art *Branch & Bound* algorithms. Actually, the algorithm is too much designed for the worst-case scenario that it is not powerful on solving common instances. The algorithm is easily generalized to solve other flowshop problems including $F2||f$ and $F3||f$, and single machine problems like $1|r_i|f$, with $f \in \{f_{max}, f_i\}$. Since it seems pretty difficult to derive even faster EEAs for $F3||C_{max}$, we prove a complexity lower bound of this problem, based on ETH, as an additional result. It shows that unless ETH fails, the $F3||C_{max}$ problem cannot be solved in subexponential time. Despite our extra research effort, we do not have much perspective on the search of a faster EEA for the $F3||C_{max}$ problem. Perhaps a wise choice is to walk out a little from the context of traditional complexity theory and try to include extra parameters into the complexity measure of algorithms, that is, try to derive parameterized algorithms.

The second contribution, presented in chapter 3, is a search tree method called *Branch & Merge* which solves the $1||\sum T_i$ problem with the time complexity converging to $\mathcal{O}^*(2^n)$ and in polynomial space. The work started by proposing a *Branch & Reduce* algorithm based on the well known Lawler's property, which decomposes the problem to two subproblems each time the job with the longest processing time is fixed to a position. This yields a structure with some similarity to *Divide & Conquer*, which can be represented as a search tree. We then analyzed the search tree carefully over the worst-case instance and found that many identical subproblems appear in different part of the tree, that is, they are generated repetitively due to Lawler's property. After a careful analysis, an operation called merging is derived. The idea is to merge all identical nodes to one whenever possible. More concretely, at each branching, a part of the search tree is developed, keeping the space

usage polynomial, and all identical nodes are identified and merged to one. Notice that we are not just cutting dominated nodes but also put the resulting node of merging in a specific position in the tree, so that subsequent mergings become possible. The algorithm goes in depth-first hence uses only polynomial space. By a careful analysis of the running time, we show that its time complexity converges to $\mathcal{O}^*(2^n)$, depending on a constant parameter k which controls the extent of merging.

The algorithm *Branch & Merge* is implemented and tested on randomly generated instances. The effectiveness of merging is proved since *Branch & Merge* runs faster than *Branch & Reduce*. However, surprisingly, when we add other well known problem properties, notably another rule that decomposes the problem and some rules eliminating the positions to consider when branching on the longest job, *Branch & Merge* does not show any superiority over the state-of-the-art *Branch & Bound* algorithm. Further analysis and preliminary experimentation revealed that the added properties have a negative effect on the efficiency of merging. In fact, for non-worst-case instances, most branching positions are already eliminated by some rules and only very few identical nodes can still be identified and merged. The corresponding gain is not larger than the extra cost caused by the merging operation itself. Nevertheless, the *Branch & Merge* framework provides an algorithmic complexity view of the solution of the $1||\sum T_i$ problem and it is generalizable to other problems benefiting similar decomposition properties, even though it seems difficult to find another scheduling problem having such property. On the improvement of practical efficiency, the work continued and finally led to the results in chapter 4.

Chapter 4 presents our third contribution, which is an extended work of chapter 3. When trying to improve the practical efficiency of *Branch & Merge*, we realized that a better way to implement the idea is to use *Memorization*, that is, instead of merging nodes in a structural way, we simply memorize the optimal solution of subproblems when they are solved at the first time, and reuse that solution whenever the same problem appears again. This is actually what has been done in the state-of-the-art *Branch & Bound* algorithm of Szwarc et al. (2001). When the *memory* is suitably implemented, the method is more efficient than *Branch & Merge* even though more memory needs to be used. *Memorization* has been implemented and tested on the $1||\sum T_i$ problem. By the finding of a new *paradox* and the implementation of a memory cleaning strategy, the method succeeded to solve instances with 300 more jobs with respect to the state-of-the-art algorithm. Then the same treatment is extended to another three problems ($1|r_i|\sum C_i$, $1|\tilde{d}_i|\sum w_i C_i$ and $F2||\sum C_i$) addressed by T'kindt et al. (2004). The four problems all together show the power of the *Memorization* paradigm when applied on sequencing problems. We call it *Branch & Memorize* to promote a systematic consideration of *Memorization* as an essential building block in branching algorithm like *Branch & Bound*. The method can certainly be used on other problems as well.

The memory cleaning strategy implemented in our *Memorization* has a surprising effect on the algorithm's efficiency. In fact, the most items contained in the *memory* are not useful at all. By cleaning these items out, the memory is used much more efficiently, which leads to an augmentation of the performance. However, it is not always easy to decide which solutions should be kept in the memory and which ones should be eliminated. As a perspective, we consider to use some *Machine Learning* techniques to make this decision. This is a so-called classification problem. Each item, corresponding to a subproblem and

its optimal solution, is a *feature* that should be suitably represented to allow possible characteristic extraction. An AI system can then be trained to judge whether a given item is worth being memorized or whether an already memorized item should be removed. Since lots of different instances are needed for a good training, each feature should also encode its corresponding instance, since an item that is useful for one instance is not necessarily equally valuable for other instances.

Another important point that we would have realized is to consider the fact that the *memory* used for solving one instance may be very useful for solving other instances. At the end of the solution of an instance, the *memory* contains a large number of already solved small problems. These results remains valid for later instances. In other words, without *Memorization*, there might be some subproblems that are solved repetitively over different instances.

At the end of this research, it is interesting and also important to review all these three results and to explore the inner link between the techniques: *Dynamic Programming*, *Branch & Merge* and *Memorization*(or *Branch & Memorize*). First of all, it worth being noticed that the conventional *Dynamic Programming* (of TSP) across all subsets can also be expressed as a search tree. Consider the search tree of the brute-force algorithm, in which the level ℓ of the tree is composed of all partial solutions of length ℓ. The number of nodes at level ℓ is hence $\binom{n}{\ell}!$. Among all these partial solutions, many consist of the same jobs (or operations, to be generic) but with a different ordering. For all partial solutions that are different orderings of the same jobs, *Dynamic Programming* allows to keep only the dominant one. This is very similar to the merging operation in *Branch & Merge* that solves the $1||\sum T_i$ problem. Note that the $1||\sum T_i$ problem can also be solved by *Dynamic Programming*, but by exploiting specific properties, *Branch & Merge* can perform the job better and only in polynomial space.

The *Dynamic Programming* algorithm can often be implemented in two ways: top-down or bottom-up along the search tree. The top-down way is a recursive implementation and the bottom-up way is actually *Memorization*. Therefore, *Dynamic Programming* and *Memorization* are also tightly related. Note that the ideas behind these two techniques are not identical: *Dynamic Programming* lays on the fact that partial solutions of an optimal solution are themselves optimal for the implied subproblems, while the idea of *Memorization* is to avoid solving a subproblem more than once, though sometimes these two ideas may coincide on a concrete problem. The link between *Branch & Merge* and *Memorization* (or *Branch & Memorize*) is pretty clear, since the latter is finally adopted in practice as a more efficient way to implement the idea of *Branch & Merge*.

All the three techniques, are superior with respect to the brute-force algorithm. This has been achieved, very reasonably, by avoiding solving identical subproblems or by eliminating dominated partial solutions. However, as introduced in section 1.2.2, there exists other algebraic techniques, such as *Inclusion & Exclusion* and *Subset Convolution* which work differently. Using these techniques to tackle scheduling problems is an interesting direction to explore, though some attempt of the author has been fruitless.

In the end, we would like to insist that *Scheduling* problems are central problems in Operation Research. It is essential to gain more knowledge about the hardness and the complexity of these problems. We hope that our works during this thesis have some positive influence on this (the publications of our works are summarized in Table 4.6). On the practical side, we also believe that EEAs have the potential to become practical, at least for moderate input size. It seems that more and more results are popping up, on the design of algorithms with better complexity for scheduling problems. This is more visible from a point of view of *Parameterized Complexity Theory*. We believe that such research will bring a new angle for solving scheduling problems and more new interesting techniques and theories will appear as a result.

Chapter	Publications
2	J. of Scheduling(Shang et al., 2017c) MISTA'15a(Shang et al., 2015) ROADEF'16(Shang et al., 2016b)
3	TCS (under review (Garraffa et al., 2017)) IPEC'17(Shang et al., 2017a) PMS'16(Shang et al., 2016a) MISTA'15b(Della Croce et al., 2015b) ROADEF'17(Shang et al., 2017b) AIRO'15(Della Croce et al., 2015a)
4	EJOR (under review (Shang et al., 2017d)) IESM'17(Shang et al., 2017e) (Best Student Paper Award, 1st place) ROADEF'17(Shang et al., 2017b)

Table 4.6: Summary of publications

Bibliography

S. Ashour. A branch-and-bound algorithm for the flow shop problem scheduling problem. *AIIE Transactions*, 2:172–176, 1970.

K. R. Baker. *Introduction to sequencing and scheduling.* John Wiley & Sons, 1974.

J. F. Bard, F. Szidarovszky, M. E. Gershon, and L. Duckstein. Techniques for Multiobjective Decision Making in Systems Management, 1988.

R. Bellman. On the theory of dynamic programming. *Proceedings of the National Academy of Sciences*, 38(8):716–719, 1952.

R. Bellman. *Dynamic Programming.* Princeton University Press, Princeton, NJ, USA, 1 edition, 1957.

R. Bellman. Dynamic programming treatment of the travelling salesman problem. *Journal of the ACM (JACM)*, 9(1):61–63, 1962.

A. Biere, M. Heule, H. Van Maaren, and T. Walsh. Conflict-driven clause learning SAT solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153, 2009.

A. Björklund and T. Husfeldt. Exact algorithms for exact satisfiability and number of perfect matchings. *Algorithmica*, 52(2):226–249, 2008.

A. Björklund, T. Husfeldt, P. Kaski, and M. Koivisto. Fourier meets Möbius: fast subset convolution. In *Proceedings of the thirty-ninth annual ACM symposium on Theory of computing*, pages 67–74. ACM, 2007.

H. L. Bodlaender and M. R. Fellows. W[2]-hardness of precedence constrained k-processor scheduling. *Operations Research Letters*, 18(2):93–97, 1995.

H. L. Bodlaender, F. V. Fomin, A. M. Koster, D. Kratsch, and D. M. Thilikos. On exact algorithms for treewidth. In *European Symposium on Algorithms*, pages 672–683. Springer, 2006.

H. L. Bodlaender, F. V. Fomin, A. M. Koster, D. Kratsch, and D. M. Thilikos. On exact algorithms for treewidth. *ACM Transactions on Algorithms (TALG)*, 9(1):12, 2012.

N. Bourgeois, B. Escoffier, V. T. Paschos, and J. M. Van Rooij. A bottom-up method and fast algorithms for max independent set. In *SWAT*, volume 6139, pages 62–73. Springer, 2010.

I. Boussaïd, J. Lepagnot, and P. Siarry. A survey on optimization metaheuristics. *Information Sciences*, 237:82 – 117, 2013.

A. Brown and Z. 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.

P. Brucker. *Scheduling Algorithms*. Springer-Verlag Berlin Heidelberg, 5th edition, 2007.

P. Brucker, A. Drexl, R. Möhring, K. Neumann, and E. Pesch. Resource-constrained project scheduling: Notation, classification, models, and methods. *European Journal of Operational Research*, 112(1):3–41, 1999.

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.

R. Carraway and T. Morin. Theory and applications of generalized dynamic programming: An overview. *Computers & Mathematics with Applications*, 16(10–11):779 – 788, 1988. ISSN 0898-1221.

R. L. Carraway, T. L. Morin, and H. Moskowitz. Generalized dynamic programming for multicriteria optimization. *European Journal of Operational Research*, 44(1):95–104, 1990.

V. Černỳ. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Journal of Optimization Theory and Applications*, 45(1):41–51, 1985.

L. S. Chandran and F. Grandoni. Refined memorization for vertex cover. *Information Processing Letters*, 93(3):125–131, 2005.

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.

C. Chu. A branch-and-bound algorithm to minimize total flow time with unequal release dates. *Naval Research Logistics (NRL)*, 39(6):859–875, 1992.

A. Cobham. The intrinsic computational difficulty of functions. In *Logic, methodology and philosophy of science, Proceedings of the 1964 International Congress*, page 24–30. North-Holland Publishing Company, 1965.

R. W. Conway, W. L. Maxwell, and L. W. Miller. *Theory of scheduling*. Addison-Wesley Educational Publishers, 1967.

S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, pages 151–158. ACM, 1971.

T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to algorithms*. MIT press Cambridge, 3 edition, 2009.

G. A. Croes. A method for solving traveling-salesman problems. *Operations Research*, 6 (6):791–812, 1958.

M. Cygan, M. Pilipczuk, M. Pilipczuk, and J. O. Wojtaszczyk. Scheduling partially ordered jobs faster than 2^n. *Algorithmica*, 68(3):692–714, 2014.

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 (TALG)*, 12(3):41, 2016.

F. Della Croce, R. Tadei, P. Baracco, and A. Grosso. A new decomposition approach for the single machine total tardiness scheduling problem. *Journal of the Operational Research Society*, pages 1101–1106, 1998.

F. Della Croce, M. Ghirardi, and R. Tadei. An improved branch-and-bound algorithm for the two machine total completion time flow shop problem. *European Journal of Operational Research*, 139(2):293–301, 2002.

F. Della Croce, M. Garraffa, L. Shang, and V. T'Kindt. A branch-and-reduce exact algorithm for the single machine total tardiness problem. 45ème Conférence de la Société Italienne de Recherche Opérationnelle (AIRO 2015), Pise (Italie), septembre 2015, 2015a.

F. Della Croce, M. Garraffa, L. Shang, and V. T'kindt. A branch-and-reduce exact algorithm for the single machine total tardiness problem. In *7th Multidisciplinary International Conference on Scheduling : Theory and Applications (MISTA 2015)*, MISTA 2015 proceedings, pages 879–881, Prague, Czech Republic, 2015b.

B. Detienne, R. Sadykov, and S. Tanaka. The two-machine flowshop total completion time problem: branch-and-bound algorithms based on network-flow formulation. *European Journal of Operational Research*, 252(3):750–760, 2016.

M. Dorigo. Optimization, learning and natural algorithms. *PhD Thesis, Politecnico di Milano, Italy*, 1992.

M. Dorigo and C. Blum. Ant colony optimization theory: A survey. *Theoretical computer science*, 344(2-3):243–278, 2005.

M. Dorigo, V. Maniezzo, A. Colorni, and V. Maniezzo. Positive feedback as a search strategy. Technical report, Technical Report No. 91-016, Politechnico di Milano, 1991.

M. Dorigo, V. Maniezzo, and A. Colorni. Ant system: optimization by a colony of cooperating agents. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 26(1):29–41, 1996.

R. G. Downey and M. R. Fellows. *Parameterized Complexity*. Monographs in Computer Science. Springer, 1999.

R. G. Downey and M. R. Fellows. *Fundamentals of parameterized complexity*, volume 4. Springer, 2013.

S. Dreyfus. Richard Bellman on the birth of dynamic programming. *Operations Research*, 50(1):48–51, 2002.

J. Du and J. Y.-T. Leung. Minimizing total tardiness on one machine is NP-hard. *Mathematics of Operations Research*, 15(3):483–495, 1990.

J. Edmonds. Paths, trees, and flowers. *Canadian Journal of Mathematics*, 17(3):449–467, 1965.

H. Emmons. One-machine sequencing to minimize certain functions of job tardiness. *Operations Research*, 17(4):701–715, 1969.

D. Eppstein. Improved algorithms for 3-coloring, 3-edge-coloring, and constraint satisfaction. In *Proceedings of the twelfth annual ACM-SIAM symposium on Discrete algorithms*, pages 329–337. Society for Industrial and Applied Mathematics, 2001.

G. W. Evans, T. L. Morin, and H. Moskowitz. Multiobjective energy generation planning under uncertainty. *IIE Transactions*, 14(3):183–192, 1982.

M. R. Fellows and C. McCartin. On the parametric complexity of schedules to minimize tardy tasks. *Theoretical computer science*, 298(2):317–324, 2003.

M. R. Fellows, S. Gaspers, and F. A. Rosamond. Parameterizing by the number of numbers. *Theory of Computing Systems*, 50(4):675–693, 2012.

J. Flum and M. Grohe. *Parameterized complexity theory*. Springer Science & Business Media, 2006.

F. V. Fomin and D. Kratsch. *Exact exponential algorithms*. Springer Science & Business Media, 2010.

F. V. Fomin, F. Grandoni, and D. Kratsch. Measure and Conquer: domination–a case study. In *International Colloquium on Automata, Languages, and Programming*, pages 191–203. Springer, 2005a.

F. V. Fomin, F. Grandoni, and D. Kratsch. Some new techniques in design and analysis of exact (exponential) algorithms. *Bulletin of the EATCS*, 87(47-77):0–288, 2005b.

F. V. Fomin, F. Grandoni, and D. Kratsch. Measure and Conquer: a simple $O(2^{0.288n})$ independent set algorithm. In *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, pages 18–25. Society for Industrial and Applied Mathematics, 2006.

F. V. Fomin, F. Grandoni, and D. Kratsch. A Measure & Conquer approach for the analysis of exact algorithms. *Journal of the ACM (JACM)*, 56(5):25, 2009.

J. M. Framinan, J. N. 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.

H. L. Gantt. *Work, wages, and profits*. Engineering Magazine Co., 1916.

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

M. R. Garey and D. S. Johnson. *Computers and intractability*. W.H. Freeman New York, 1979.

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.

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. https://hal.archives-ouvertes.fr/hal-01477835, Feb 2017.

F. Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, 1986.

F. Glover. Tabu search—part I. *ORSA Journal on Computing*, 1(3):190–206, 1989.

F. Glover. Tabu search—part II. *ORSA Journal on Computing*, 2(1):4–32, 1990.

D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

R. L. Graham, E. L. Lawler, J. K. Lenstra, and A. R. Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of Discrete Mathematics*, 5:287–326, 1979.

R. L. Graham, D. E. Knuth, and O. Patashnik. *Concrete Mathematics: A Foundation for Computer Science*. Addison Wesley, 2 edition, 1994.

J. A. Gromicho, J. J. V. Hoorn, F. S. da Gama, and G. T. Timmer. Solving the job-shop scheduling problem optimally by Dynamic Programming. *Computers & Operations Research*, 39(12):2968–2977, 2012.

Y. Gurevich and S. Shelah. Expected computation time for Hamiltonian path problem. *SIAM Journal on Computing*, 16(3):486–502, 1987.

I. Harjunkoski, C. Maravelias, P. Bongers, P. Castro, S. Engell, I. Grossmann, J. Hooker, C. Méndez, G. Sand, and J. Wassick. Scope for industrial applications of production scheduling models and solution methods. *Computers & Chemical Engineering*, 62:161 – 193, 2014.

M. Held and R. M. Karp. A dynamic programming approach to sequencing problems. *Journal of the Society for Industrial & Applied Mathematics*, 10(1):196–210, 1962.

M. I. Henig. Vector-valued dynamic programming. *SIAM Journal on Control and Optimization*, 21(3):490–499, 1983.

E. Horowitz and S. Sahni. Computing partitions with applications to the knapsack problem. *Journal of the ACM (JACM)*, 21(2):277–292, 1974.

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.

R. Impagliazzo and R. Paturi. On the Complexity of k-SAT. *Journal of Computer and System Sciences*, 62(2):367 – 375, 2001.

K. Jansen, S. Kratsch, D. Marx, and I. Schlotter. Bin packing with fixed number of bins revisited. *Journal of Computer and System Sciences*, 79(1):39–49, 2013a.

K. Jansen, F. Land, and K. Land. Bounding the Running Time of Algorithms for Scheduling and Packing Problems. In F. Dehne, R. Solis-Oba, and J.-R. Sack, editors, *Algorithms and Data Structures*, volume 8037 of *Lecture Notes in Computer Science*, pages 439–450. Springer Berlin Heidelberg, 2013b.

T. Jian. An $o(2^{0.304n})$ algorithm for solving maximum independent set problem. *IEEE Transactions on Computers*, 100(9):847–851, 1986.

S. Johnson. Optimal two-and three-stage production schedules with setup times included. *Naval Research Logistics (NRL)*, 1(1):61–68, 1954.

A. Jouglet, P. Baptiste, and J. Carlier. Branch-and-Bound Algorithms for TotalWeighted Tardiness. In *Handbook of scheduling: Algorithms, models, and performance analysis*. Chapman and Hall/CRC, 2004.

R. M. Karp. Reducibility among combinatorial problems. In *Complexity of Computer Computations*, pages 85–103. Springer, 1972.

R. M. Karp. Dynamic programming meets the principle of inclusion and exclusion. *Operations Research Letters*, 1(2):49–51, 1982.

J. E. Kelley Jr and M. R. Walker. Critical-path planning and scheduling. In *Papers presented at the December 1-3, 1959, eastern joint IRE-AIEE-ACM computer conference*, pages 160–173. ACM, 1959.

S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by Simulated Annealing. *Science*, 220(4598):671–680, 1983.

J. Kneis, A. Langer, and P. Rossmanith. A fine-grained analysis of a simple independent set algorithm. In *LIPIcs-Leibniz International Proceedings in Informatics*, volume 4. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

A. Kooli and M. Serairi. A mixed integer programming approach for the single machine problem with unequal release dates. *Computers & Operations Research*, 51:323–330, 2014.

C. Koulamas. The single-machine total tardiness scheduling problem: review and extensions. *European Journal of Operational Research*, 202(1):1–7, 2010.

H.-T. Kung, F. Luccio, and F. P. Preparata. On finding the maxima of a set of vectors. *Journal of the ACM (JACM)*, 22(4):469–476, 1975.

T. Ladhari and M. Haouari. A computational study of the permutation flow shop problem based on a tight lower bound. *Computers & Operations Research*, 32(7):1831–1847, 2005.

B. Lageweg, J. Lenstra, and A. Rinnooy Kan. A general bounding scheme for the permutation flow-shop problem. *Operations Research*, 26(1):53–67, 1978.

E. L. Lawler. A "pseudopolynomial" algorithm for sequencing jobs to minimize total tardiness. *Annals of Discrete Mathematics*, 1:331–342, 1977.

C.-Y. Lee. Machine scheduling with an availability constraint. *Journal of Global Optimization*, 9(3-4):395–416, 1996.

C.-Y. Lee. Machine scheduling with availability constraints. In L. JY-T, editor, *Handbook of Scheduling: Algorithms, Models and Performance Analysis*, chapter 22, pages 22.1–22.13. Chapman & Hall/CRC, Boca Raton, FL, 2004.

J. J. K. Lenstra. The mystical power of twoness: in memoriam Eugene L. Lawler. *Journal of Scheduling*, 1(1):3, 1998.

C. Lenté, M. Liedloff, A. Soukhal, and V. T'Kindt. On an extension of the *Sort & Search* method with application to scheduling theory. *Theoretical Computer Science*, 511:13–22, 2013.

C. Lenté, M. Liedloff, A. Soukhal, and V. T'Kindt. Exponential Algorithms for Scheduling Problems. *HAL*, 2014. URL `https://hal.archives-ouvertes.fr/hal-00944382`.

J. Y. Leung. *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.

D. Lokshtanov and J. Nederlof. Saving space by algebraization. In *Proceedings of the forty-second ACM symposium on Theory of computing*, pages 321–330. ACM, 2010.

Z. 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.

G. McMahon and P. Burton. Flow-shop scheduling with the branch-and-bound method. *Operations Research*, 15(3):473–481, 1967.

L. G. Mitten. Preference order dynamic programming. *Management Science*, 21(1):43–46, 1974.

M. Mnich and A. Wiese. Scheduling and fixed-parameter tractability. *Mathematical Programming*, 154(1-2):533–562, 2015.

P.-A. Morin. Algorithmes exponentiels en ordonnancement. Projet recherche & développement, Ecole Polytechnique de l'Université François Rabelais de Tours, Tours, France, 2014-2015.

J. F. Muth and G. L. Thompson. *Industrial scheduling*. Prentice-Hall, 1963.

T. Nicholson. A sequential method for discrete optimization problems and its application to the assignment, travelling salesman, and three machine scheduling problems. *IMA Journal of Applied Mathematics*, 3(4):362–375, 1967.

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.

M. Pinedo. *Scheduling (Third Edition)*. Springer, 2008.

M. E. Posner. Minimizing weighted completion times with deadlines. *Operations Research*, 33(3):562–574, 1985.

C. Potts. An adaptive branching rule for the permutation flow-shop problem. *European Journal of Operational Research*, 5(1):19–25, 1980.

C. Potts and L. Van Wassenhove. A decomposition algorithm for the single machine total tardiness problem. *Operations Research Letters*, 1(5):177–181, 1982.

C. Potts and L. Van Wassenhove. Integrating scheduling with batching and lot-sizing: a review of algorithms and complexity. *Journal of the Operational Research Society*, pages 395–406, 1992.

C. N. Potts and M. Y. Kovalyov. Scheduling with batching: A review. *European Journal of Operational Research*, 120(2):228–249, 2000.

C. N. Potts and V. A. Strusevich. Fifty years of scheduling: a survey of milestones. *Journal of the Operational Research Society*, 60(1):S41–S68, 2009.

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.

K. Pruhs, J. Sgall, and E. Torng. Online scheduling. In L. JY-T, editor, *Handbook of Scheduling: Algorithms, Models and Performance Analysis*, chapter 15, pages 15.1–15.43. Chapman & Hall/CRC, Boca Raton, FL, 2004.

S. Reza Hejazi and S. Saghafian. Flowshop-scheduling problems with makespan criterion: a review. *International Journal of Production Research*, 43(14):2895–2929, 2005.

J. M. Robson. Algorithms for maximum independent sets. *Journal of Algorithms*, 7(3): 425–440, 1986.

K. H. Rosen. *Discrete mathematics and its applications*. McGraw-Hill, 7 edition, 2012.

M. Rosenman and J. S. Gero. Pareto optimal serial dynamic programming. *Engineering Optimization*, 6(4):177–183, 1983.

R. Ruiz and C. Maroto. A comprehensive review and evaluation of permutation flowshop heuristics. *European Journal of Operational Research*, 165(2):479 – 494, 2005. Project Management and Scheduling.

E. Sanlaville and G. Schmidt. Machine scheduling with availability constraints. *Acta Informatica*, 35(9):795–811, 1998.

G. Schmidt. Scheduling with limited machine availability. *European Journal of Operational Research*, 121(1):1–15, 2000.

R. Schroeppel and A. Shamir. A $T = O(2^{n/2})$, $S = O(2^{n/4})$ algorithm for certain NP-Complete Problems. *SIAM journal on Computing*, 10(3):456–464, 1981.

L. Shang, C. Lenté, M. Liedloff, and V. T'Kindt. An exponential dynamic programming algorithm for the 3-machine flowshop scheduling problem to minimize the makespan. In *7th Multidisciplinary International Conference on Scheduling : Theory and Applications (MISTA 2015)*, MISTA 2015 proceedings, pages 755–758, Prague, Czech Republic, Aug. 2015.

L. Shang, M. Garraffa, F. Della Croce, and V. T'Kindt. An Exact Exponential Branch-and-Merge Algorithm for the Single Machine Total Tardiness Problem. In *16th International Conference on Project Management and Scheduling (PMS 16)*, PMS 2016 proceedings, pages 182–185, Valencia, Espagne, 2016a.

L. Shang, C. Lenté, M. Liedloff, and V. T'Kindt. Programmation dynamique exponentielle pour des problèmes d'ordonnancement de type flowshop à 3 machines. 17ème congrés de la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF'16), Compiègne, 10 au 12 février 2016, 2016b.

L. Shang, M. Garraffa, F. Della Croce, and V. T'Kindt. Merging nodes in search trees: an exact exponential algorithm for the single machine total tardiness scheduling problem. In *12th International Symposium on Parameterized and Exact Computation (IPEC 2017)*, volume 89 of *LIPIcs*, pages 28:1–28:12, Vienna, Austria, Aug 2017a. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik.

L. Shang, M. Garraffa, F. Della Croce, and V. T'Kindt. An Exact Node-Merging Algorithm for the Single Machine Total Tardiness Problem : Experimental Considerations. 18ème congrés de la Société Française de Recherche Opérationnelle et d'Aide à la Décision (ROADEF'17), Metz, 22 au 24 février 2017, 2017b.

L. Shang, C. Lenté, M. Liedloff, and V. T'Kindt. Exact exponential algorithms for 3-machine flowshop scheduling problems. *Journal of Scheduling*, pages 1–7, 2017c.

L. Shang, V. T 'kindt, and F. Della Croce. The Memorization Paradigm: Branch & Memorize Algorithms for the Efficient Solution of Sequencing Problems. https://hal.archives-ouvertes.fr/hal-01599835, Oct 2017d.

L. Shang, V. T'Kindt, and F. Della Croce. Exact Solution of the Single Machine Total Tardiness Problem : the Power of Memorization. In *7th International Conference on Industrial Engineering and System Management (IESM 2017)*, pages 268–272, Saarbrucken, Germany, Oct 2017e. Best Student Paper Award.

M. Sipser. The history and status of the p versus np question. In *Proceedings of the twenty-fourth annual ACM symposium on Theory of computing*, pages 603–618. ACM, 1992.

C. Smutnicki. Some results of the worst-case analysis for flow shop scheduling. *European Journal of Operational Research*, 109(1):66 – 87, 1998.

M. Sviridenko. A Note on Permutation Flow Shop Problem. *Annals of Operations Research*, 129(1):247–252, 2004.

W. Szwarc. Single machine total tardiness problem revisited. *Creative and Innovative Approaches to the Science of Management, Quorum Books*, pages 407–419, 1993.

W. Szwarc and S. K. Mukhopadhyay. Decomposition of the single machine total tardiness problem. *Operations Research Letters*, 19(5):243–250, 1996.

W. Szwarc, F. Della Croce, and A. Grosso. Solution of the single machine total tardiness problem. *Journal of Scheduling*, 2(2):55–71, 1999.

W. Szwarc, A. Grosso, and F. D. Croce. Algorithmic paradoxes of the single-machine total tardiness problem. *Journal of Scheduling*, 4(2):93–104, 2001.

R. E. Tarjan and A. E. Trojanowski. Finding a maximum independent set. *SIAM Journal on Computing*, 6(3):537–546, 1977.

V. T'kindt and J.-C. Billaut. *Multicriteria scheduling: theory, models and algorithms*. Springer Science & Business Media, 2006.

V. T'kindt, F. Della Croce, and C. Esswein. Revisiting branch and bound search strategies for machine scheduling problems. *Journal of Scheduling*, 7(6):429–440, 2004.

R. Van Bevern. On the parameterized complexity of scheduling with side constraints: Recent results and new challenges. In *Proceedings of the 13th Workshop on Models and Algorithms for Planning and Scheduling Problems*, pages 119–121, 2017.

R. Van Bevern and A. V. Pyatkin. Completing partial schedules for open shop with unit processing times and routing. In *Computer Science – Theory and Applications: 11th International Computer Science Symposium in Russia, CSR 2016, St. Petersburg, Russia, June 9-13, 2016, Proceedings*, pages 73–87, Cham, 2016. Springer International Publishing.

R. Van Bevern, R. Bredereck, L. Bulteau, C. Komusiewicz, N. Talmon, and G. J. Woeginger. Precedence-constrained scheduling problems parameterized by partial order width. In *Proceedings of the 9th DOOR, volume 9869 of LNCS*, pages 105–120. Springer International Publishing, 2016.

R. Van Bevern, R. Niedermeier, and O. Suchỳ. A parameterized complexity view on non-preemptively scheduling interval-constrained jobs: few machines, small looseness, and small slack. *Journal of Scheduling*, 20(3):255–265, 2017.

B. Villarreal and M. H. Karwan. An interactive dynamic programming approach to multicriteria discrete programming. *Journal of Mathematical Analysis and Applications*, 81 (2):524–544, 1981.

G. J. Woeginger. Exact Algorithms for NP-hard Problems: A Survey. In M. Jünger, G. Reinelt, and G. Rinaldi, editors, *Combinatorial Optimization — Eureka, You Shrink!*, volume 2570 of *Lecture Notes in Computer Science*, pages 185–207. Springer Berlin Heidelberg, 2003.

M. Xiao and H. Nagamochi. Exact algorithms for maximum independent set. *Information and Computation*, 255(1):126–146, 2017.

L. Zhang, C. F. Madigan, M. H. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*, pages 279–285. IEEE Press, 2001.

Résumé :

Cette thèse synthétise les travaux de recherches réalisés pendant les études doctorales de l'auteur. L'objectif de ces travaux est de proposer des algorithmes exacts qui ont une meilleure complexité, temporelle ou spatiale, dans le pire des cas pour des problèmes d'ordonnancement qui sont $\mathcal{NP}$-difficiles. En plus, on s'intéresse aussi à évaluer leurs performances en pratique.

Trois contributions principales sont rapportées. La première concerne un algorithme du type *Dynamic Programming* qui résout le problème $F3||C_{max}$ en $\mathcal{O}^*(3^n)$ en temps et en espace. L'algorithme est généralisé facilement à d'autres problèmes du type Flowshop, y compris les problèmes $F2||f$ et $F3||f$, et aux problèmes d'ordonnancement à une seule machine tels que les problèmes $1|r_i|f$, avec $f \in \{f_{max}, f_i\}$.

La seconde contribution porte sur l'élaboration d'une méthode arborescente appelée *Branch & Merge* pour résoudre le problème $1||\sum T_i$ en $\mathcal{O}^*((2+\epsilon)^n)$ en temps avec $\epsilon > 0$ arbitrairement petit et en espace polynomial. Le travail se base sur l'observation que de nombreux sous-problèmes identiques apparaissent répétitivement pendant la résolution du problème global. A partir de ça, une opération appelée *merge* est proposée, qui fusionne les sous-problèmes (les noeuds dans l'arbre de recherche) identiques autant que possible. Cette méthode doit pouvoir être généralisée à d'autres problèmes.

Le but de la troisième contribution est d'améliorer les performances en pratique des algorithmes exacts procédant par parcours d'un arbre de recherche. D'abord nous avons aperçu qu'une meilleure façon d'implémenter l'idée de *Branch & Merge* est d'utiliser une technique appelée *Memorization*. Avec la découverte d'un nouveau *paradoxe* et la mis en place d'une stratégie de nettoyage de mémoire, notre algorithme a résolu les instances qui ont 300 tâches de plus par rapport à l'algorithme de référence pour le problème $1||\sum T_i$. Avec ce succès, nous avons testé *Memorization* sur trois autres problèmes d'ordonnancement notés $1|r_i|\sum C_i$, $1|\tilde{d}_i|\sum w_i C_i$ et $F2||\sum C_i$, précédemment traités par T'kindt et al. (2004). Les résultats finaux des quatre problèmes ont montré la puissance de *Memorization* appliquée aux problèmes d'ordonnancement. Nous nommons ce paradigme *Branch & Memorize* afin de promouvoir la considération systématique de l'intégration de *Memorization* dans les algorithmes de branchement comme *Branch & Bound*, en tant qu'un composant essentiel. La méthode peut aussi être généralisée pour résoudre d'autres problèmes qui ne sont pas forcément des problèmes d'ordonnancement.

Mots clés :

Algorithmes exponentiels, ordonnancement, brancher et réduire, branch and merge, mémorisation, branch and memorize, programmation dynamique, flowshop, somme de retards

Abstract :

This thesis summarizes the author's PhD research works on the design of exact algorithms that provide a worst-case (time or space) guarantee for $\mathcal{NP}$-hard scheduling problems. Both theoretical and practical aspects are considered with three main results reported.

The first one is about a *Dynamic Programming* algorithm which solves the $F3||C_{max}$ problem in $\mathcal{O}^*(3^n)$ time and space. The algorithm is easily generalized to other flowshop problems including $F2||f$ and $F3||f$, and single machine scheduling problems like $1|r_i|f$, with $f \in \{f_{max}, f_i\}$.

The second contribution is about a search tree method called *Branch & Merge* which solves the $1||\sum T_i$ problem with the time complexity converging to $\mathcal{O}^*(2^n)$ and in polynomial space. The work is based on the observation that many identical subproblems appear during the solution of the input problem. An operation called *merge* is then derived, which merges all identical nodes to one whenever possible and hence yields a better complexity.

Our third contributionaims to improve the practical efficiency of exact search tree algorithms solving scheduling problems. First we realized that a better way to implement the idea of *Branch & Merge* is to use a technique called *Memorization*. By the finding of a new *algorithmic paradox* and the implementation of a memory cleaning strategy, the method succeeded to solve instances with 300 more jobs with respect to the state-of-the-art algorithm for the $1||\sum T_i$ problem. Then the treatment is extended to another three problems $1|r_i|\sum C_i$, $1|\tilde{d}_i|\sum w_i C_i$ and $F2||\sum C_i$ previously addressed by T'kindt et al. (2004). The results of the four problems all together show the power of the *Memorization* paradigm when applied on sequencing problems. We name it *Branch & Memorize* to promote a systematic consideration of *Memorization* as an essential building block in branching algorithms like *Branch & Bound*. The method can surely also be used to solve other problems, which are not necessarily scheduling problems.

Keywords :

Exact exponential algorithms, scheduling, branch and reduce, branch and merge, memorization, branch and memorize, dynamic programming, flowshop, total tardiness