Efficient algorithms to solve scheduling problems with a variety of optimization criteria

Thèse

Hamed Fahimi

Doctorat en informatique
Philosophiæ doctor (Ph.D.)

Québec, Canada

© Hamed Fahimi, 2016

Efficient algorithms to solve scheduling problems with a variety of optimization criteria

Thèse

Hamed Fahimi

Sous la direction de:

Claude-Guy Quimper, directeur de recherche

Résumé

La programmation par contraintes est une technique puissante pour résoudre, entre autres, des problèmes d'ordonnancement de grande envergure. L'ordonnancement vise à allouer dans le temps des tâches à des ressources. Lors de son exécution, une tâche consomme une ressource à un taux constant. Généralement, on cherche à optimiser une fonction objectif telle la durée totale d'un ordonnancement. Résoudre un problème d'ordonnancement signifie trouver quand chaque tâche doit débuter et quelle ressource doit l'exécuter. La plupart des problèmes d'ordonnancement sont NP-Difficiles. Conséquemment, il n'existe aucun algorithme connu capable de les résoudre en temps polynomial. Cependant, il existe des spécialisations aux problèmes d'ordonnancement qui ne sont pas NP-Complet. Ces problèmes peuvent être résolus en temps polynomial en utilisant des algorithmes qui leur sont propres. Notre objectif est d'explorer ces algorithmes d'ordonnancement dans plusieurs contextes variés. Les techniques de filtrage ont beaucoup évolué dans les dernières années en ordonnancement basé sur les contraintes. La proéminence des algorithmes de filtrage repose sur leur habilité à réduire l'arbre de recherche en excluant les valeurs des domaines qui ne participent pas à des solutions au problème. Nous proposons des améliorations et présentons des algorithmes de filtrage plus efficaces pour résoudre des problèmes classiques d'ordonnancement. De plus, nous présentons des adaptations de techniques de filtrage pour le cas où les tâches peuvent être retardées. Nous considérons aussi différentes propriétés de problèmes industriels et résolvons plus efficacement des problèmes où le critère d'optimisation n'est pas nécessairement le moment où la dernière tâche se termine. Par exemple, nous présentons des algorithmes à temps polynomial pour le cas où la quantité de ressources fluctue dans le temps, ou quand le coût d'exécuter une tâche au temps t dépend de t.

Abstract

Constraint programming is a powerful methodology to solve large scale and practical scheduling problems. Resource-constrained scheduling deals with temporal allocation of a variety of tasks to a set of resources, where the tasks consume a certain amount of resource during their execution. Ordinarily, a desired objective function such as the total length of a feasible schedule, called the makespan, is optimized in scheduling problems. Solving the scheduling problem is equivalent to finding out when each task starts and which resource executes it. In general, the scheduling problems are NP-Hard. Consequently, there exists no known algorithm that can solve the problem by executing a polynomial number of instructions. Nonetheless, there exist specializations for scheduling problems that are not NP-Complete. Such problems can be solved in polynomial time using dedicated algorithms. We tackle such algorithms for scheduling problems in a variety of contexts. Filtering techniques are being developed and improved over the past years in constraint-based scheduling. The prominency of filtering algorithms lies on their power to shrink the search tree by excluding values from the domains which do not yield a feasible solution. We propose improvements and present faster filtering algorithms for classical scheduling problems. Furthermore, we establish the adaptions of filtering techniques to the case that the tasks can be delayed. We also consider distinct properties of industrial scheduling problems and solve more efficiently the scheduling problems whose optimization criteria is not necessarily the makespan. For instance, we present polynomial time algorithms for the case that the amount of available resources fluctuates over time, or when the cost of executing a task at time t is dependent on t.

Contents

Résumé iii

Abstract iv

Contents v

List of Tables viii

List of Figures ix

Aknowledgments xii

Introduction 1

1 Constraint satisfaction and constraint programming 6
- 1.1 Combinatorial optimization . . . 6
- 1.2 Constraint satisfaction problems . . . 6
  - 1.2.1 What is a CSP? . . . 7
- 1.3 Constraint programming . . . 8
  - 1.3.1 Searching solutions . . . 8
  - 1.3.2 Supports and local consistency . . . 9
  - 1.3.3 Filtering algorithms and constraint propagation . . . 10

2 Scheduling Theory 12
- 2.1 Scheduling framework . . . 12
  - 2.1.1 Terminology and representation . . . 12
  - 2.1.2 Objective Functions . . . 14
  - 2.1.3 A family of scheduling problems . . . 15
  - 2.1.4 Classifying scheduling in terms of resource and task types . . . 15
  - 2.1.5 Three-field characterization . . . 16
- 2.2 Global Constraints Used in Scheduling . . . 17
  - 2.2.1 ALL-DIFFERENT constraint . . . 17
  - 2.2.2 Global Cardinality . . . 17
  - 2.2.3 INTER-DISTANCE . . . 18
  - 2.2.4 MULTI-INTER-DISTANCE . . . 18
  - 2.2.5 Disjunctive constraint . . . 18
  - 2.2.6 CUMULATIVE constraint . . . 19
- 2.3 Polynomial time algorithms for scheduling problems . . . 19

2.3.1 Related Work 20
2.3.2 Scheduling Graph 20
2.3.3 Network Flows 22

3 Filtering algorithms for the disjunctive and cumulative constraints 24
3.1 Overload Checking 25
3.2 Time-Tabling 28
3.3 Edge-Finding 29
3.3.1 $(\Theta - \Lambda)_L-$tree 31
3.4 Extended-Edge-Finding 32
3.5 Not-First/Not-Last 33
3.6 Energetic Reasoning 34
3.7 Detectable Precedences 35
3.8 Precedence Graph 35
3.9 Combination of filtering techniques 36
3.10 Filtering techniques in the state of the art schedulers 36

4 Linear time Algorithms for Disjunctive constraint 38
4.1 Preliminaries 39
4.1.1 Union-Find 39
4.2 Time-Tabling 40
4.3 The Time Line Data Structure 42
4.4 Overload Checking 45
4.5 Detectable Precedences 45
4.6 Experimental Results 47
Two sorting algorithms 48
Filtering algorithms 50
4.7 Minimizing maximum lateness and total delay 50
4.8 Conclusion 52

5 Overload Checking and Edge-Finding for Robust Cumulative Scheduling 53
5.1 Preliminaries and the general framework 53
5.1.1 Robust cumulative constraint 54
5.2 Robust Overload Checking 55
5.2.1 The general form of robust Overload Checking rule 55
5.2.2 Robust earliest energy envelope 56
5.2.3 Robust Overload Checking rule in terms of robust energy envelope 56
5.2.4 Θ^r_L-tree 57
5.2.5 Robust Overload Checking algorithm 58
5.3 Robust Edge-Finding 59
5.3.1 Filtering the earliest starting times 61
Robust Edge-Finding rule for filtering the earliest starting times 61
Robust $\Lambda-$earliest energy envelope 62
$(\Theta - \Lambda)^r_L-$tree 62
Robust Edge-Finding algorithm for filtering the earliest starting times 63
5.3.2 Filtering the latest completion times 71
Robust Edge-Finding rules for filtering the latest completion times 71
Robust latest energy envelope 72

Robust $\Lambda-$latest energy envelope 73
$(\Theta - \Lambda)^r_U-$tree 73
Robust Edge-Finding algorithm for filtering the latest completion times . 75
5.4 Experiments 82
5.5 Conclusion 84

6 Variants of Multi-Resource Scheduling Problems with Equal Processing Times 85
6.1 Variety of machine numbers through the time 86
6.2 General Objective Function 88
6.3 Scheduling problems with monotonic objective functions 89
6.4 Scheduling problems with periodic objective functions 90
6.4.1 Scheduling problem as a network flow 90
6.4.2 Periodic objective function formulated as a network flow 92
6.5 Minimizing maximum lateness 96
6.6 Conclusion 96

Conclusion 97

Bibliography 101

List of Tables

2.1 Resource environment possibilities depending on the value of α. 16
2.2 Task possibilities depending on the value of β. 16
2.3 Objective possibilities depending on the value of γ. 16

3.1 A set of tasks for which neither Overload Checking fails, nor is there a feasible solution. 25
3.2 The information of a set of tasks $\mathcal{I} = \{A, B, C, D\}$ with a resource of capacity $\mathcal{C} = 3$. 32

4.1 The information of a set of tasks $\mathcal{I} = \{A, B, C\}$. 42
4.2 Comparison of Θ-tree and time line. 43
4.3 Parameters of the tasks used to illustrate the scheduling process on the time line. . . . 44
4.4 Open-shop with n jobs and m tasks per job. Ratio of the cumulative number of backtracks between all instances of size $n \times m$ after 10 minutes of computations. OC: our Overload Checking vs. Vilím's. DP: our Detectable Precedences vs Vilím's. TT: Our Time-Tabling vs Ouellet et al. All algorithms use insertion sort. 51
4.5 Job-shop with n jobs and m tasks per job. Ratio of the cumulative number of backtracks between all instances of size $n \times m$ after 10 minutes of computations. OC: our Overload Checking vs. Vilím's. DP: our Detectable Precedences vs Vilím's. TT: Our Time-Tabling vs Ouellet et al. All algorithms use insertion sort. 51
4.6 Random instances with n tasks. Times are reported in milliseconds. Algorithms implementing the same filtering technique lead to the same number of backtracks (bt). TT: $\Theta-$tree, TL: time line, UF:Union-Find data structure. 51

5.1 A set of tasks $\mathcal{I} = \{A, B, C, D, E\}$ to execute on a resource of capacity $\mathcal{C} = 4$. The Overload Checking fails according to (5.7) for $\Theta = \{A, B, C, D\} \subseteq \mathcal{I}, \Theta^0 = \{A^0, C^0\}$ and $\Theta^1 = \{B^1, D^1\}$. 55
5.2 A set of tasks $\mathcal{I} = \{A, B, C, D\}$ to execute on a resource of capacity $\mathcal{C} = 7$. 62
5.3 The *prec* array which is obtained after the execution of the algorithm 11 on the instance of example 5.3.1 67
5.4 The *prec* array which is obtained after the execution of the algorithm 15 on the instance of example 5.3.2 78

6.1 Summary of the contributions mentioned in this thesis. NA stands for no previously known algorithms. 100

List of Figures

1.1 The search tree corresponding to a CSP with $X = \{X_1, X_2, X_3\}$, $\text{dom}(X_1) = \{1, 2, 3\}$, $\text{dom}(X_2) = \{0, 1\}$, $\text{dom}(X_3) = \{1\}$ and $C : X_1 \geq 2X_2 + X_3$. At the red node, the search backtracks, as the assignments on the path to that node violate the constraint C. The yellow node indicates an unvisited node, due to the occurrence of a backtrack on the branch connection leading to that node. 9

2.1 The components of a task A with $\text{est}_A = 0, \text{lct}_A = 33, p_A = 16, c_A = 6$, carrying out on a resource with $\mathcal{C} = 11$. A has $\text{e}_A = 6 \times 16 = 96$ units of energy. 13

2.2 For the set of tasks $\mathcal{I}$= {A,B,C,D,E,F,G} which must execute on a resource of capacity $\mathcal{C} = 8$ the assignment $(s_A, s_B, s_C, s_D, s_E, s_F, s_G) = (0, 4, 8, 19, 18, 20, 28)$ provides a valid schedule, where $\text{dom}(s_A) = [0, 10), \text{dom}(s_B) = [2, 8), \text{dom}(s_C) = [5, 21), \text{dom}(s_D) = [15, 33), \text{dom}(s_E) = [13, 26), \text{dom}(s_F) = [17, 31), \text{dom}(s_G) = [20, 33)$. 14

2.3 The scheduling graph with five tasks with processing times $p = 2$. 22

2.4 A network flow and a residual network flow. 23

3.1 Overload Checking triggers a failure for $\Theta = \{A, B, C, D\}$. 25

3.2 The cumulative Θ_L-tree corresponding to $\mathcal{I} = \{A, B, C, D\}$ with a resource of capacity $\mathcal{C} = 3$. 27

3.3 The compulsory part of the task A with $\text{est}_A = 0, \text{lct}_A = 10, p_A = 6$ is [4,6). 28

3.4 The precedence $\Theta_1 = \{B, E\} \prec D$ gives rise to filtering $\text{est}_D = 20$ and the precedences $D \prec \Theta_2 = \{A\}$ and $B \prec \Theta_3 = \{A, D, E\}$ give rise to $\text{lct}_D = 40, \text{lct}_B = 25$. 30

3.5 Detectable precedences detects $A \ll C, B \ll C, A \ll D, B \ll D$ for the tasks of table 3.5. 35

3.6 The appropriate sequence for the DISJUNCTIVE constraint, presented in Vilím's thesis [88], which achieves the fixpoint 37

4.1 A trace of Time-Tabling algorithm for the tasks of table 4.1. 42

4.2 The tasks $\mathcal{I}_{\text{ect}} = \{1, 2, 3, 4\}$ and the visual representation of a solution to the DISJUNCTIVE constraint. The algorithm Detectable Precedences prunes the earliest starting times $\text{est}'_3 = 19$ and $\text{est}'_4 = 13$ 48

4.3 Implementing insertion sort (x axis) and quick sort (y axis) when the state of the art is used for the instances of open shop problem. The figures represent the logarithmic based scale of the number of backtracks occurred within 10 minutes. OC stands for Overload Checking, DP stands for Detectable Precedences and TT stands for Time-Tabling. 49

4.4 Implementing insertion sort (x axis) and quick sort (y axis) when our algorithms are used for the instances of open shop problem. The figures represent the logarithmic based scale of the number of backtracks occurred within 10 minutes. OC stands for Overload Checking, DP stands for Detectable Precedences and TT stands for Time-Tabling. 49

5.1 In the picture (a), C^0 and C^1 are scheduled in the Θ_L^r-tree. The coloured nodes in blue represent the affected nodes during the update of the tree. In this case, $\Theta^0 = \{C^0\}$ and $\Theta^1 = \{C^1\}$. In the picture (b), A^0 is scheduled in the Θ_L^r-tree, but not A^1. In this case, $\Theta^0 = \{C^0, A^0\}$ and $\Theta^1 = \{C^1\}$. In the picture (c), A^1 and D are scheduled. In this case, $\Theta^0 = \{C^0, A^0, D^0\}$ and $\Theta^1 = \{C^1, A^1\}$. Picture (d) represents the status of the Θ_L^r-tree when processing $t = 25$. Since $\text{Env}^2(\Theta^0, \Theta^1) = \text{Env}^2_{\text{root}} = 102 > 4 \cdot 25 = 100$ the Overload Checking triggers a failure. 60
5.2 In the picture (a), the algorithm starts with a full $(\Theta - \Lambda)_L^r-$tree. In the picture (b), C^1 gets unscheduled from Θ^1 and scheduled in Λ^1. The modifications to Θ and Λ sets are respectively coloured in green and blue. In the picture (c), C^0 is unscheduled from Θ^0 and scheduled in Λ^0. In the picture (d) after C^1 is found as the responsible tasks, it gets unscheduled from Λ^1. In the picture (e), D^1 gets unscheduled from Θ^1 and scheduled in Λ^1. 66
5.3 The state of $\Theta_L^{cr}-$tree in the iteration that est_C gets adjusted. The blue arrows show the path which is traversed by the algorithm 12. This algorithm locates the yellow node. The algorithm 13 starts from the yellow node and the green arrows show the path traversed by this algorithm to the root. 70
5.4 In the figure a, the algorithm starts with a full $(\Theta - \Lambda)_U^r-$tree. In the figure b, A gets unscheduled from Θ and scheduled in Λ. In the figure c, A^1 and then A^0 are detected as responsible tasks and get unscheduled. In the figure d, B is unscheduled. 77
5.5 The logarithmic scale graphs as the results of running Time-Tabling and Overload Checking (denoted OC-TT) or Time-Tabling and Edge-Finding(denoted EF-TT) with lexicographic heuristics in terms of backtrack numbers as well as elapsed times. The horizontal axis corresponds to the Time-Tabling and the vertical axis corresponds to Overload Checking or Edge-Finding. 82
5.6 The logarithmic scale graphs as the results of running Time-Tabling and Overload Checking (denoted OC-TT) or Time-Tabling and Edge-Finding(denoted EF-TT) with impact based search heuristics in terms of backtrack numbers as well as elapsed times. The horizontal axis corresponds to the Time-Tabling and the vertical axis corresponds to Overload Checking or Edge-Finding. 83
5.7 The logarithmic scale graphs as the results of running Time-Tabling and Overload Checking (denoted OC-TT) or Time-Tabling and Edge-Finding(denoted EF-TT) with domoverwdeg heuristics in terms of backtrack numbers as well as elapsed times. The horizontal axis corresponds to the Time-Tabling and the vertical axis corresponds to Overload Checking or Edge-Finding. 84

6.1 A trace of the algorithm 19, where $p = 2$ and $T = [(1, 3), (3, 5), (4, 2), (6, 6)]$. The sequences T' and T'' are constructed, as represented at lines (b) and (c). 88
6.2 A network flow with 5 tasks. The cost on the forward, backward, and null edges are written in black. These edges have unlimited capacities. The capacities of the nodes from the source and to the sinks are written in blue. These edges have a null cost. . . 91

6.3 The compressed version of the graph on Figure 6.2. The numbers over the edges connected to the source and sinks stand for the capacities and the costs are omitted. . 93

Aknowledgments

During the preparation of this work, I had the pleasure to greatly benefit from thoughtful comments given by my supervisor Claude-Guy Quimper, a source of acumen and encouragement, who accompanied the development of this work with numerous advices for improvements and shared his useful insights in this effort. This work would have not been in such a state if he did not provide me with his fruitful cooperation.

The author is also indebted to the community members of this dissertation, François Laviolette and Jonathan Gaudreault for supporting this work by many inspired discussions, not to mention Thierry Petit and Richard Khoury for reading and commenting my manuscript as well as evaluating the defence.

Thanks are also due to anonymous reviewers for their useful recommendations to improve the manuscripts before publication.

I also wish to acknowledge Giovanni Won Dias Baldini Victoret for his assistance in the experimental results of chapter 4 and Alban Derrien for providing his implementation of the Time-Tabling algorithm for the results of chapter 5.

I owe a debt of gratitude to my wife. This dissertation is dedicated to my beloved parents.

Introduction

Scheduling is an occasionally used concept to which we are accustomed in our daily life, notwithstanding keeping a standard definition for that is often overlooked. Normally, we desire to plan the activities that we are engaged in beforehand for purposes such as time-saving, reducing cost charges, etc. For instance, suppose you are invited to attend the graduation ceremony of a group of students in the school, which is scheduled to start at a specified date. The place where the party takes place is located in downtown and you have to arrange for your departure in accordance with the public transportation schedule so that you arrive to the party on time. Such an occasion simply demonstrates a scheduling problem for which you need to care for two *schedule*s, the ceremony as well as the public transportation, and these schedules are planned in advance.

Since the late 1950's and early 1960's *scheduling problems* have attracted a large amount of research [7, 93]. Covered by a wide variety of disciplines such as computer science (CS), artificial Intelligence (AI), operational research (OR), engineering, manufacturing, management, maintenance, etc., scheduling problems are mostly known as an interdisciplinary field of study, as they arise in numerous application settings in the real-world.

The general class of scheduling problems that we consider in this dissertation involves a group of activities to be carried out over a set of resources. The activities require certain amounts of resources to process. The activities and resources can have different interpretations, depending on the context. Moreover, most of the scheduling problems are optimization problems. That is, there is a desired objective to be attained, such as minimizing the duration of the schedule, minimizing the costs, maximizing the profits, etc. Frequently, there are given precedences among the activities, prescribing the order in which they must be processed. The basic form of a schedule usually establishes the dates at which the activities should start such that the precedence and the alternative constraints hold and the objective function is optimized. Providing such dates to acquire a valid schedule for the activities is equivalent to finding a solution for the problem. This framework for the scheduling problems is expressive enough to capture dozens of features which arise in practice.

The input size of the general scheduling problems is typically proportional to the number of activities, the number of resources and the number of bits to represent the largest integer among the components of an activity. Unfortunately, the problem of solving most of the scheduling problems is NP-hard [29]. In fact, not only solving, but also verifying whether a feasible solution exists can take exponential

effort. That is why relatively less attempts have been made to achieve polynomial time algorithms for the scheduling problems, thus far.

In the literature, there are a variety of areas of general methods used to solve scheduling problems such as MIP (mixed integer programming), satisfiability testing (SAT), logic-based Benders decomposition (LBBD) [36, 38] and constraint programming (CP). We focus on the latter method. Constraint programming is a generic purpose paradigm to solve combinatorial problems by interpreting them in terms of constraints. A typical constraint satisfaction problem (CSP) consists of a set of decision variables, to each member of which a domain is associated. The domain includes the valid values that are assignable to the variables. Furthermore, a set of constraints that circumscribe the decision variables are established and a single objective function is to be optimized. Many problems can be presented in terms of constraints. Furthermore, many disciplines such as OR and AI, have developed methods for satisfying constraints. In particular, *constraint-based scheduling* provides powerful search strategies to solve scheduling problems, by taking advantage of constraint programming. Among the successful application areas of constraint programming are the *cumulative* and *disjunctive* scheduling. Cumulative scheduling differs from disjunctive scheduling in the sense that in cumulative scheduling several activities are allowed to run simultaneously. Each activity consumes a certain amount of resource, and the CUMULATIVE constraint ensures that the accumulation of resource usage by the activities underway at any time does not overflow the resource. In the disjunctive scheduling no more than one activity can execute on a resource and the DISJUNCTIVE constraint ensures that the activities do not overlap at any time.

The main goal of this dissertation is to address special structures of the scheduling problems encountered in the industry, from a constraint-based viewpoint. Furthermore, we aim at tackling distinct properties of industrial scheduling problems with different objective functions that can be encountered in practical applications. Constraints can be used to encode all sort of these problems.

The domains of variables in a CSP include values which are not consistent with some constraints of the problem. Removing all inconsistencies from the domain of variables with respect to the CUMULATIVE constraint is NP-Hard. However, several rules are proposed which can partially remove inconsistencies in polynomial time. The scheduling problems that we consider to a great extent rely on a problem solving paradigm from constraint programming which is called *constraint propagation*. This method is a reduction technique which detects inconsistencies through the repetitive analysis of the variables and makes the problem simpler to solve. The constraint propagation process uses several *filtering algorithms*. These algorithms discover time zones in which the activities cannot start. Discovering such time zones prevent the solver from exploring some portions of the search space where no feasible solutions lie. Since constraint programming is based upon filtering algorithms, devising efficient algorithms is essential and this objective has captured the interest of many researchers in the CP community. This dissertation makes contributions to this research area by developing efficient, effective and fast filtering algorithms to solve conventional scheduling problems.

Scheduling environments are not always static in the real world and uncertainty is prevalent in this context. For instance, the operations might take longer than expected to execute, a supply chain for resources can break down and the resources becomes unavailable, etc. Such disruptions unavoidably cause delays in the activities. On the other hand, practically, it is not possible to re-compute the solutions in such cases. Therefore, it is of crucial importance to develop filtering algorithms which deal with such environments. An approach is to maintain a robust schedule that absorbs some level of unforeseen events when at most a certain number of activities are delayed. Even though this is not mentioned as an objective function, robustness is a desired criterion in a schedule.

Although the scheduling problems are NP-complete in the general form, specializations to the properties of the problem can yield to problems that can be solved in polynomial time. For instance, if all the activities execute with the same duration over multiple resources, it is possible to find a solution. Furthermore a common objective function is to minimize the makespan, i.e. when the last task finishes. However, in practice, alternative objective functions often occur depending on the circumstances and context and yet it is essential that managers operate their business with the utmost efficiency. This dissertation particularly aims at solving more efficiently scheduling problems whose optimization criteria is not necessarily the makespan. For instance, we consider the cases such as when the amount of available resources fluctuates over time with respect to the activities. We also elaborate on specific objective functions that give rise to polynomiality.

The reminder of the monograph on hand consists of 7 chapters.

Chapter 1 introduces the standard CSP and the principal concepts which are relevant to the context of scheduling problems in this dissertation from a constraint based viewpoint. Moreover, a couple of overly simple, but illustrative examples are provided in order to the elucidate the notions.

Chapter 2 starts with a description of the basic entities that contribute to the construction of a general scheduling problem, including activities, resources and objective functions. After introducing these concepts, a hands-on application of this framework is described. Then, a family of well known scheduling problems are introduced. Afterwards, the scheduling problems are classified depending on the resource and activity types. Also, a systematic notation in order to refer to particular scheduling problems is presented. Furthermore, six global constraints which can be interpreted in terms of scheduling problems are introduced. Finally, this chapter introduces a class of particular scheduling problems which are solvable in polynomial time.

Over the past two decades, a variety of standard constraint propagation algorithms have been introduced for the scheduling problems. These principles are frequently employed in order to reduce the search space for the scheduling problems that are NP-hard. The basic algorithms are as follows. Overload Checking, Time-Tabling, Edge-Finding , Extended-Edge-Finding, Not-First/Not-Last, Energetic Reasoning, Detectable Precedences and the Precedence Graph. Chapter 3 surveys these well-known filtering techniques for the DISJUNCTIVE and CUMULATIVE constraint and introduces the state of the art algorithms.

Chapters 4, 5 and 6 compose three main chapters of this book which reflect three contributions.

Chapter 4 is devoted to the DISJUNCTIVE constraint, which is extensively used in the industrial applications, such as manufacturing and supply chain management [37]. This constraint simply ensures that two activities A and B which execute on the same resource do not overlap in time. Propagation of the DISJUNCTIVE constraint ensures that either A precedes B or B precedes A. Chapter 4 presents three new and efficient filtering algorithms that all have a linear running time complexity in the number of activities. The first algorithm filters the activities according to the rules of Time-Tabling. The second algorithm performs an Overload Checking that could also be adapted for the CUMULATIVE constraint. The third algorithm enforces the rules of Detectable Precedences. The two last algorithms use a new data structure that we introduce and that we call the *time line*. This data structure provides constant time operations that were previously implemented in logarithmic time by a data structure which is called Θ-tree. While the state of the art algorithms admit a running time of $O(n \log(n))$, the proposed new algorithm are linear. Accordingly, the new algorithms run faster as the size of the input increases. The experimental results verify that these new algorithms are competitive even for a small number of activities and outperform existing algorithms as the number of activities increases. Moreover, it turns out that the proposed time line data structure is powerful enough to solve more efficiently particular scheduling problems for which polynomial time algorithms exist. We published this work in the proceedings of the 28th AAAI Conference on Artificial Intelligence (AAAI-14) [26].

Chapter 5 is concerned with scheduling in robust cumulative contexts. It develops two new filtering algorithms for the FLEXC constraint, a constraint that models cumulative scheduling problems where up to r out of n activities can be delayed while keeping the schedule valid. By extending the $\Theta-$tree, which is used in the state of the art algorithms for Overload Checking and Edge-Finding, theses filtering rules for this framework are adapted. It turns out that the complexities of the state of the art algorithms for these techniques are maintained, when the number of delayed activities r is constant. The experimental results verify a stronger filtering for these methods when used in conjunction with Time-Tabling. Furthermore, the computation times show a faster filtering for specific heuristics for quite a lot of instances. [1]

In Chapter 6 a particular scheduling problem which can be solved in polynomial time is examined. This is the problem of scheduling of a set of activities, all of which have equal processing times, to be executed without interruption over multiple resources with given release time and deadlines. We consider the case that the number of resources fluctuates over time and we present a polynomial time algorithm for this problem. Further, we consider different objective functions for this problem. For instance, the assumption that the execution of every activity at any time incurs a cost has received little attention in the literature. We show that if the cost is a function of the activity and the time, the objective of minimizing the costs yields an NP-Hard problem. Further, we specialize this objective function to the case that it is merely contingent on the time and show that although this case is pseudo-polynomial in time, one can derive polynomial algorithms for the problem, provided the cost function

[1] This work is planned to be shortly submitted to the journal of *Constraints* while this dissertation is under evaluation.

is monotonic or periodic. Finally, we point out how polynomial time algorithms can be adapted with the objective of minimizing maximum lateness. We published this work in the proceedings of the 9th Annual International Conference on Combinatorial Optimization and Applications (COCOA 2015) [27].

This work finishes with a summary in Chapter 7.

Chapter 1

Constraint satisfaction and constraint programming

1.1 Combinatorial optimization

Combinatorial optimization is a branch of study in applied mathematics as well as computer science whose focus is primarily to solve optimization problems. Ordinarily, the goal of a combinatorial optimization problem is to find a feasible solution over a finite and discrete structure subject to an objective function to be optimized. The following simply exemplifies a combinatorial optimization problem.

Example 1.1.1. Suppose that m employees are required to run errands on n different positions in a factory, each one of which takes $p_j, 1 \leq j \leq n$, units of time and an employee cannot run two errands simultaneously. The objective is to minimize the total finishing time.

An inductive approach employs a generate-and-test method by enumerating all feasible solutions to find the optimal solution. Nonetheless, this approach mostly does not yield a solution within reasonable time limits for growing size of inputs, as the set of possibilities grows exponentially fast which makes it exhaustive to consider the set in-depth. There exist alternative techniques such as dynamic programming, branch and bound, etc. to solve combinatorial optimization problems. After all, they commonly all suffer from a huge size of search space as the problem size increases. Next sections describe a powerful methodology to quickly search through an enormous space of possibilities and reduce the computational effort required for solving combinatorial optimization problems.

1.2 Constraint satisfaction problems

Constraint satisfaction problems (CSP) have emerged as a major area of focus for AI community researchers over the past decades. The historical roots of constraint programming (CP) can be traced

back to 1960s and 1970s, where the paradigm first arose in artificial intelligence and computer graphics [81]. After it was gradually realized that a host of complex and practical problems in applied sciences could be interpreted in terms of satisfaction problems, CSP evolved into a rather mature field. Covering a large spectrum of real-world applications, such as artificial intelligence, database systems, programming languages, graphical interfaces, natural language processing and operations research, nowadays constraint programming provides a versatile tool and powerful technique to solve combinatorial optimization problems. Notably, one can recast a variety of paradigms arising in AI, including scheduling, timetabling, resource allocation, planning, assignment problems and maximum flows in terms of a CSP [69, 23].

1.2.1 What is a CSP?

Let us formally go over the details of CSP. A *variable* is a symbol to which different values could be assigned. The set of candidate values to be assigned to a variable determines the *domain* of the variable. In this dissertation, we are only concerned with constraint satisfaction problems with finite domains. A *constraint* can be regarded as a restriction established on the variable assignments. Formally, a constraint C is a logical relation defined on a set of variables.

Roughly speaking, *a constraint satisfaction problem (CSP)* is a mathematical problem defined on a set of variables, each one with a finite and discrete domain subject to certain constraints.

Definition 1.2.1. An instance of a CSP is composed of the sets

$$X = \{X_1, ..., X_n\}, D = \{\text{dom}(X_1), ..., \text{dom}(X_n)\}, C = \{C_1, ..., C_m\}, X' = \{X_{C_1}, ..., X_{C_m}\}$$

where each $X_i \in X$, $1 \leq i \leq n$, is a variable with the finite domain $\text{dom}(X_i) \in D$ and each $C_j \in C$, $1 \leq j \leq m$, is a constraint defined over the set $X_{C_j} \in X'$ such that $X_{C_j} \subseteq X$. That is, the variables in X_{C_j} are chosen from the universal set of variables X. X_{C_j} is called the *scope* of C_j and the cardinality of X_{C_j}, denoted $|X_{C_j}|$, is called the *arity* of C_j.

A *global constraint* is a constraint that correlates a non-fixed number of variables with each other. Global constraints are a more general form of a constraint, that can be expressed with simpler constraints of fixed arity. Although they may be expressed as a conjunction of further constraints, they frequently simplify the model of a problem and facilitate the work of solvers by providing a concise and expressive manner of modelling a condition. The advantage of global constraints is that they provide a specialized filtering algorithm that prunes the domains of the variables much more than the conjunction of elementary constraints. As an example, the ALL-DIFFERENT$(X_1, ..., X_n)$ is a global constraint which associates pairwise distinct values to the variables $X_1, ..., X_n$. It turns out that the ALL-DIFFERENT constraint filters more than $O(n^2)$ constraints of pairwise inequalities $X_i \neq X_j$ for $1 \leq i, j \leq n$ [66]. The number of variables in the scope of a global constraint can take any value. That is, the arity of a global constraint is a parameter of that constraint.

Definition 1.2.2. A *solution* to a CSP instance is an assignment of values to the variables from their domains which satisfies all of the constraints of the CSP.

Typically, the goal of a CSP is to find one or all of the solutions.

Example 1.2.1. Consider a CSP instance with $X = \{X_1, X_2, ..., X_8\}$, $D = \{\text{dom}(X_1), \text{dom}(X_2), ..., \text{dom}(X_8)\}$ and $C = \{C_1, C_2, C_3\}$, where
$C_1 : X_1^2 < X_2^2$;
$C_2 : X_3 + X_4 = X_5$;
$C_3 : (X_6 \neq X_7) \wedge (X_7 \neq X_8) \wedge (X_6 \neq X_8)$ (*i.e.* X_6, X_7 and X_8 simultaneously take different values)

and

$$\begin{array}{lll} \text{dom}(X_1) = \{1, 2, 3, 4, 5, 6\} & \text{dom}(X_2) = \{0, 1, 2, 3, 4\} & \text{dom}(X_3) = \{0, 1, 3\} \\ \text{dom}(X_4) = \{0, 1, 2, 5\} & \text{dom}(X_5) = \{0, 2, 3, 6\} & \text{dom}(X_6) = \{4, 6\} \\ \text{dom}(X_7) = \{4, 6\} & \text{dom}(X_8) = \{4, 5, 6\} & \end{array}$$

The tuple (3,4,1,3,4,9,2,1) provides a solution for this CSP. □

1.3 Constraint programming

CSPs are intractable and belong to the class of NP-complete problems [80]. Accordingly, much efforts are made to diminish the elapsed time required to solve CSPs. *Constraint programming* is a technique which provides such a prospect. Constraint programming languages offer built-in constraints which make it easy to model a problem into a CSP. Thus far, there are multiple toolkits and packages designed for developing constraint-based systems, such as CHIP [22], Choco [60], Gecode [31], Comet [35], etc. Our experiments presented in this dissertation were implemented via Choco solver, which is an open source CP library in Java.

1.3.1 Searching solutions

A *search tree* is a tree whose internal nodes represent the partial assignments and its leaves represent the candidate solutions. The root of the tree is initialized with the empty set, to which no variable is associated. Figure 1.1 represents the search tree corresponding to the CSP $X_1 \geq 2X_2 + X_3$ with $\text{dom}(X_1) = \{1, 2, 3\}$, $\text{dom}(X_2) = \{0, 1\}$ and $\text{dom}(X_3) = \{1\}$.

The solver traverses the search tree to seek a solution for a CSP. In the following we present a well known type of search in constraint programming.

Depth-first search (DFS) is a general technique to visit the nodes of a search tree in a preorder traversal. That is, the algorithm starts at the root of the tree and explores all of the nodes by continuing down

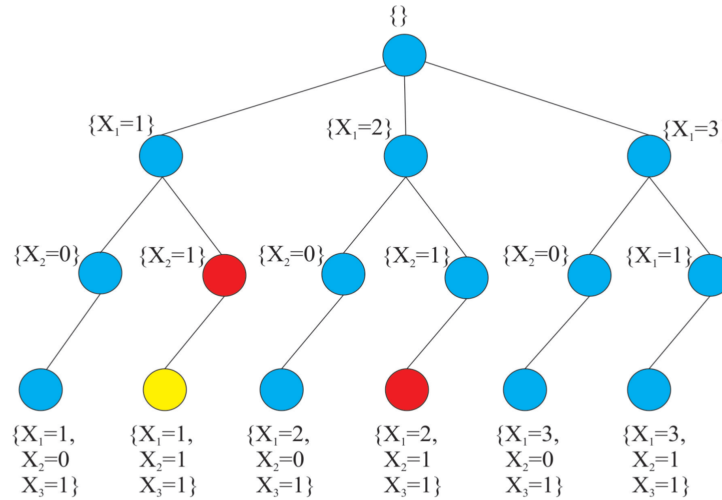


Figure 1.1 – The search tree corresponding to a CSP with $X = \{X_1, X_2, X_3\}, \text{dom}(X_1) = \{1, 2, 3\}, \text{dom}(X_2) = \{0, 1\}$, $\text{dom}(X_3) = \{1\}$ and $C : X_1 \geq 2X_2 + X_3$. At the red node, the search backtracks, as the assignments on the path to that node violate the constraint C. The yellow node indicates an unvisited node, due to the occurrence of a backtrack on the branch connection leading to that node.

the tree so that each node is visited before its children. *Backtracking* is a type of search where the traversal of a sub-tree is interrupted when the violation of a constraint is detected in the partial solution of a node. A backtrack rules out the last assignment and proceeds to another assignment by bringing the solver back to the parent node. For instance, since the assignments on the path leading to the red nodes of the search tree in figure 1.1 violate the constraint C, these assignments are ruled out and a backtrack is triggered. A backtrack prevents the solver to visit the yellow node. If all domains are reduced to a single value, a feasible solution is achieved.

Choosing the appropriate order in which the variables get instantiated greatly affects the elapsed time to find a solution. Indeed, without a good ordering, it can take too long to solve even moderate-sized CSPs. A *branching heuristic* is a policy choice made on branching over the search tree by giving priority to certain solutions.

1.3.2 Supports and local consistency

A particularity of constraint programming is its ability to prune the search tree. Such a reduction of the search space leads to faster computation times.

In the following, the notation $t[X]$ for the tuple t that satisfies the constraint refers to the value assigned to the variable X by t.

Definition 1.3.1. A solution for the constraint C is called a *support*. If t is a support such that $t[X_i] = v$ for $v \in \text{dom}(X_i)$, t is said to be a support for v.

Definition 1.3.2. Given a constraint C over a set $X = \{X_1, ..., X_n\}$,

(i) A tuple t that satisfies C is a *domain support* if for $1 \leq i \leq n$ and $X_i \in X$, $t[X_i] \in \text{dom}(X_i)$.
(ii) A tuple t that satisfies C is an *interval support* if for $1 \leq i \leq n$ and $X_i \in X$, $\min(\text{dom}(X_i)) \leq t[X_i] \leq \max(\text{dom}(X_i))$.

A domain support provides a solution for the constraint. Moreover, a domain support is also an interval support. In contrast, the opposite does not necessarily hold.

Example 1.3.1. In the example 1.2.1 with regard to the constraint C_2, $(1, 5, 6)$ is a domain support and $(3, 2, 5)$ is an interval support, but not a domain support.

Over the past few decades, there have been developments in introducing different concepts of *consistency* in order to identify the variable assignments which are not consistent and must be ruled out.

Definition 1.3.3. A constraint C is
(i) *bounds consistent* if for each $X_l \in X_C, 1 \leq l \leq |X_C|$, there is an interval support in C for each of the values min(dom(X_l)) and max(dom(X_l));
(ii) *range consistent* if for each $X_l \in X_C$, there is an interval support in C for each value $a \in$ dom(X_l);
(iii) *domain consistent* if for each $X_l \in X_C$, there is a domain support in C for each value $a \in$ dom(X_l).

It can be inferred from the definition 1.3.3 that enforcing domain consistency removes all values from the domains that do not have a domain support. Enforcing bounds consistency removes the smallest and greatest values from the domains until the constraint becomes bounds consistent and enforcing range consistency removes all values from the domains that do not have an interval support.

Notice that each constraint has its own algorithm that enforces a given level of consistency.

Example 1.3.2. In the example 1.2.1, with regard to the constraint C_3, the value $4 \in$ dom(X_8) has the interval support (6,5,4) and does not have a domain support. Hence, C_3 is not domain consistent, albeit it is range consistent.

1.3.3 Filtering algorithms and constraint propagation

Initially, the domains of a CSP define a search space that could be exponential in size. They may include values which are not consistent with all or some constraints of the problem. To reduce the search space, the solvers use *filtering algorithms*. These algorithms are associated to the constraints and they keep on excluding values of the domains that do not lead to a feasible solution and therefore are not part of any solution. The algorithms are invoked over and over until no further pruning can occur. A filtering algorithm is said to be *idempotent* if applying it multiple times does not result in more filtering.

Example 1.3.3. Consider the CSP instance from the example 1.2.1. Since for all $i \in \{4, 5, 6\} \subseteq \text{dom}(X_1)$ and all $j \in \text{dom}(X_2)$, $i^2 \geq j^2$, these values of i do not satisfy C_1, thereby they do not lead to a solution and are dropped from $\text{dom}(X_1)$ during the filtering process. Likewise, the values $j \in \{0, 1\} \subseteq \text{dom}(X_2)$ do not lead to a solution. Thus, the filtering updates the domains to $\text{dom}(X_1) = \{1, 2, 3\}$ and $\text{dom}_{X_2} = \{2, 3, 4\}$.

Constraint propagation is the mechanism of determining how the constraints and domains of variables interact. One can filter a constraint C_1 so that it becomes consistent. Then, one can filter a constraint C_2 so that it becomes consistent, too. However, if C_1 and C_2 share variables in their scope, there is no more guarantee that the constraint C_1 is still consistent. Therefore, C_1 must be filtered again. Keeping track of which constraint is consistent and calling the filtering algorithms of constraints that could be inconsistent is the constraint propagation process. Constraint propagation is an elementary technique to accelerate the search of a solution, which can considerably reduce the search space. It basically simplifies the search by detecting inconsistencies which are captured by the constraints of the problem through the repetitive analysis of the variables and exploiting redundant constraints which are discovered during the search. Notice that constraint propagation stops when no more inconsistencies for the values of the domains can be detected. In such a state the CSP becomes *locally consistent*.

Mackworth [51] proposed AC3 / GAC3 which is an example of a well-known algorithm for constraint propagation. Viewing the CSP as a directed graph whose nodes and edges respectively represent the variables and the constraints which circumscribe the variables, this algorithms iterates over each edge e of the graph and eliminates the values from the domains which do not satisfy the constraints associated to e. The algorithm maintains a set of edges by adding all of the edges pointing to the filtered value except e after the removal of a value.

There exist stronger consistencies than local consistency, such as path consistency [54] and singleton consistency [18, 19].

Chapter 2

Scheduling Theory

Project scheduling problems deal with the temporal allocation of a variety of activities to a set of resources over time in order to achieve some objectives. Since this concept can be interpreted quite broadly, a multitude of practical problems arising in diverse areas such as transportation, distribution settings, manufacturing environments, etc. fit within this framework. Notably, the scheduling problems are challenging combinatorial optimization problems.

Since the early days of operations research, scheduling problems have been intensively investigated by the OR and AI community [7, 93, 70, 77]. This scenario can be considered from different mathematical points of view. In this dissertation, we take it into consideration from a constraint-based standpoint.

2.1 Scheduling framework

In order to have a proper statement of the scheduling problems which are most frequently used throughout the forthcoming chapters, this section describes the basic terminology, notations and different types of the scheduling problems.

2.1.1 Terminology and representation

A traditional scheduling problem is formulated by a triple $(\mathcal{R}, \mathcal{I}, \gamma)$, where $m \geq 1$ shared *resources* (or *machines*) $\mathcal{R} = \{R_1, ..., R_m\}$ are required to process a set of n *tasks* (or *activities*) $\mathcal{I} = \{1, ..., n\}$ such that a desired objective function γ is attained. In the context of this dissertation, each $R_j \in R, 1 \leq j \leq m,$ is capable of performing each task $i \in \mathcal{I}$, equally and all the resources are identical. If a task i has access to all resources of R whatsoever, the resources are called *parallel*. The resources and tasks in the problem vary depending on their associated organization. Presuming the time is discrete, that is the values of the attributes are integers, we establish the following conventions in order to characterize each task $i \in \mathcal{I}$.

The *release time* or the *earliest starting time* of i, denoted est_i, is the earliest date at which i becomes

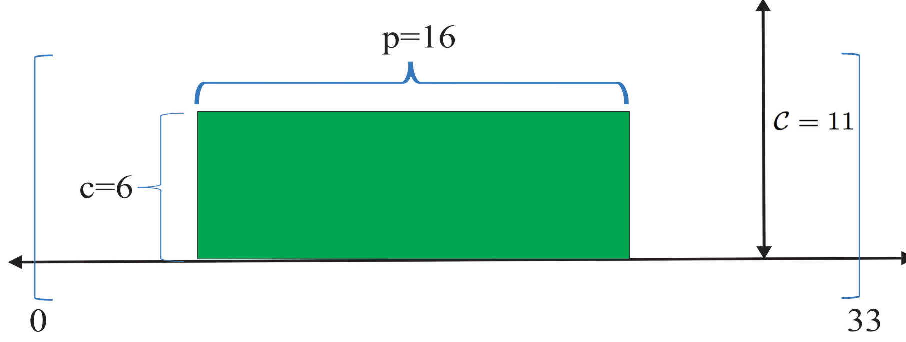


Figure 2.1 – The components of a task A with $\text{est}_A = 0, \text{lct}_A = 33, p_A = 16, c_A = 6$, carrying out on a resource with $\mathcal{C} = 11$. A has $\text{e}_A = 6 \times 16 = 96$ units of energy.

available to be executed on any resource. The *deadline* or the *latest completion time* of i, denoted lct_i, is the latest date at which i can cease to execute on any resource. The *duration* or *processing time* of i, denoted p_i, is the total elapsed time if i executes on any resource. The *latest starting time* (lst_i) of a task is the maximum date at which it can start executing and the *earliest completion time* (ect_i) of a task is the minimum date at which it can cease to execute. These two values are computed with $\text{lst}_i = \text{lct}_i - p_i$ and $\text{ect}_i = \text{est}_i + p_i$, respectively. The *missing date* of i, denoted o_i, is the earliest time point by starting at which i oversteps its deadline. This component is computed by $o_i = \text{lst}_i + 1$.

Throughout this dissertation, we focus on deterministic scheduling problems, where all attributes of the problem defined above are given with certainty in advance.

Resource-constrained scheduling is typically dedicated to the problems dealing with tasks that use a constant amount of resource during their execution. A fixed and positive value $\mathcal{C}$, called the *capacity* or *height*, is associated to a resource and each task i consumes a certain amount c_i of this capacity during its execution. Moreover, the sum of the capacities of the tasks executing at a time t should not exceed $\mathcal{C}$. The *energy* of a task i, denoted e_i, is the amount of resource that is consumed by i during its execution and it is computed by $\text{e}_i = c_i p_i$. Figure 2.1 illustrates a task together with its associated data, to be executed on a resource of capacity $\mathcal{C} = 11$.

One may generalize the notations defined above to an arbitrary subset $\Theta \subseteq \mathcal{I}$ of tasks as follows

$$\begin{aligned}
\text{est}_\Theta &= \min\{\text{est}_i : i \in \Theta\} \\
\text{lct}_\Theta &= \max\{\text{lct}_i : i \in \Theta\} \\
o_\Theta &= \max\{o_i : i \in \Theta\} \\
p_\Theta &= \sum_{i \in \Theta} p_i \\
\text{e}_\Theta &= \sum_{i \in \Theta} \text{e}_i
\end{aligned}$$

For an empty set, we assume that $\text{lct}_\emptyset = -\infty$, $o_\emptyset = \text{est}_\emptyset = \infty$ and $p_\emptyset = \text{e}_\emptyset = 0$.

The *starting time* of a task, denoted S_i, is the time point at which it starts executing. The *ending time*

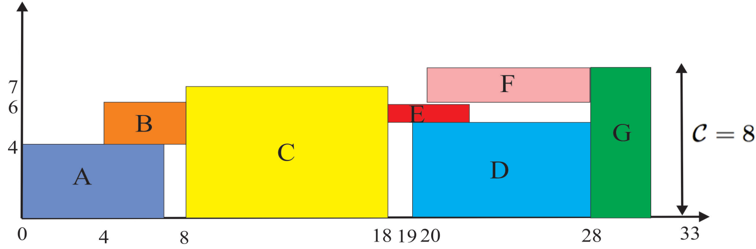


Figure 2.2 – For the set of tasks $\mathcal{I}$= {A,B,C,D,E,F,G} which must execute on a resource of capacity $\mathcal{C} = 8$ the assignment $(s_A, s_B, s_C, s_D, s_E, s_F, s_G) = (0, 4, 8, 19, 18, 20, 28)$ provides a valid schedule, where $\text{dom}(s_A) = [0, 10), \text{dom}(s_B) = [2, 8), \text{dom}(s_C) = [5, 21), \text{dom}(s_D) = [15, 33), \text{dom}(s_E) = [13, 26), \text{dom}(s_F) = [17, 31), \text{dom}(s_G) = [20, 33)$.

of a task, denoted E_i, is the time point at which it ceases to execute. Since in general the goal of a scheduling problem is to achieve a consistent schedule, the starting times of the tasks are the decision variables of the problem to be chosen from $[\text{est}_i, \text{lst}_i]$ for every task i and finding out feasible values for the starting times yields a solution for the problem. When several tasks concurrently compete for the same resource, finding out an optimal solution for the problem is commonly difficult. If a solution is found, the assignment of starting times to the tasks provides a *schedue* for the problem. Figure 2.2 demonstrates a schedule for a set of tasks.

Notice that a task i executes at time t, if $S_i \leq t < S_i + p_i$. Moreover, since $\text{est}_i \leq S_i < o_i$, therefore $E_i \leq \text{lct}_i$.

Some criteria such as the scarcity of resources limits the usage of tasks independently. Typically, in the scheduling problems there are given precedence constraints between the tasks $i, j \in \mathcal{I}$, prescribing the order in which they must be carried out. If the task i precedes j, the precedence constraint between i and j implies that $S_j \geq E_i$.

2.1.2 Objective Functions

Numerous objectives can be considered in scheduling problems. A frequent objective is to finish the tasks as soon as possible or formally, to minimize the total length of a feasible schedule, called the *makespan* and denoted $E_{\max}$. In fact, the makespan of a schedule provides the latest ending time. The idea behind this motivation is that the sooner a schedule is completed, the more alternative tasks can be executed due to the freedom of resources, which spontaneously reduces the penalties which could have been caused by overstepping the deadlines. The *due date* of i, denoted $\bar{d}_i$, is the date at which i is due to be completed without incurring a cost. The *lateness* of a task is a measure of how much the completion of a task exceeds its due date. This value is computed with $L_i = E_i - \bar{d}_i$. Minimizing the maximum lateness, denoted $L_{\max}$ is an alternative objective, occasionally considered in the literature. The idea behind such objective is that the tasks incur a cost proportional to the deviation between their ending time and due dates and the goal is to minimize such costs. If executing a task i at time t costs $w(i, t)$, *minimizing costs per task and per time* aims at minimizing the sum of costs, i.e. $\sum_{i,t} w(i, S_i)$.

If executing any task at time t costs $w(t)$, *minimizing task costs per time* aims at minimizing $\sum_i w(S_i)$.

Example 2.1.1. (taken from [23]) In an airline terminal at a large international airport there are several parking positions or terminal gates accommodating the arrival and departure of hundreds of aircrafts according to a certain schedule per day. The aircrafts should be assigned to appropriate gates that are available at the respective arrival times. In this scenario, the gates are considered as resources. The gates are scarce resources indeed, and it is important to use them as efficiently as possible. A gate can be occupied by one aircraft at each time. During this period, the tasks execute for certain minimum units of time. These tasks include, for example, handling and servicing the aircrafts by the personnel, loading and unloading the baggage, boarding the passengers, etc. The tasks are subject to the precedence constraints. The starting time of a task is determined by the arrival of the aircraft which is dependent on the flight schedule. The completion time of a task is determined by the departure of the aircraft. The objective can be considered to minimize the maximum lateness of the aircraft.

In the next section, we introduce a family of well-known scheduling problems which are extensively studied in the literature.

2.1.3 A family of scheduling problems

The family of *shop scheduling* problems is composed of *job shop*, *open shop* and *flow shop* problems. In these problems, $\mathcal{I}$ is clustered into sets of *job*s. As a common property for these problems, the jobs belonging to $\mathcal{I}$ ought to execute on the resources of $\mathcal{R}$ and each resource can run at most one job at the same time. In the *job shop* problem the jobs of each task have their own order of execution on the resources. In the *flow shop* problem, the order of execution of jobs within the tasks are identical for each task. In the *open shop* problem the order of execution of jobs on the resources is immaterial. Ordinarily, in the shop scheduling problems the objective is to minimize the makespan.

2.1.4 Classifying scheduling in terms of resource and task types

One can categorize the major scheduling problems by the resource configuration. A scheduling problem for which each resource can execute at most one task at each time is said to be *disjunctive*. In the disjunctive scheduling, $c_i = \mathcal{C}$ for $i \in \mathcal{I}$. The family of shop scheduling problems fall into this category. The scheduling problems in which several tasks can run on a resource, provided the capacity of the resource does not exceed is said to be *cumulative*.

Depending on the type of tasks found in the problem, we distinguish *non-preemptive scheduling* and *preemptive scheduling*. In non-preemptive scheduling, the tasks are not allowed to be interrupted. That is, each task must execute without interruption ever since it starts executing until it finishes. In preemptive scheduling, the tasks can be interrupted during their execution and resumed possibly on another resource. Notice that in non-preemptive scheduling, the constraint $S_i + p_i = E_i$ holds and in preemptive scheduling, the constraint $S_i + p_i \leq E_i$ holds.

2.1.5 Three-field characterization

The wide variety of problem types motivated Graham et al. [33] to introduce a systematic notation for referring to a scheduling problem. Such a characterization is a three-field notation $\alpha|\beta|\gamma$, where α denotes the resource environment, β refers to the job characteristics and γ provides the objective function to be optimized. Table 2.1 characterizes the resource environment possibilities depending on the value of α.

α	**Environment Description**
1	$\|\mathcal{R}\| = 1$, i.e. there is a single resource.
P	The resources are identical and parallel.
Q	The resources are parallel, but not identical.
R	Each job might have a specific processing time on each resource.
Om	Open shop problem.
Jm	Job shop problem.
Fm	Flow shop problem.

Table 2.1 – Resource environment possibilities depending on the value of α.

The notation β describes additional properties or constraints for the tasks. The table 2.2 characterizes these possibilities depending on the value of β.

β	**Task Description**
pmtn	The tasks are preemptive.
prec	Precedence constraints exist among the tasks.
$\mathrm{est}_i, \mathrm{lct}_i, \bar{d}_i$	If either or some of these symbols are present, the problem takes such variables as input.
$p_i = a$	If this symbol is present, then all of the tasks have processing times equal to a.

Table 2.2 – Task possibilities depending on the value of β.

In addition to finding a feasible schedule, one usually wants to optimize an objective function γ. Table 2.3 characterizes more possibilities depending on the value of γ.

γ	**Objective function**
$\sum_i E_i$	Minimizing the sum of the completion times
$E_{\max}$	Minimizing makespan
$L_{\max}$	Minimizing maximium lateness
$\sum_i (E_i - \mathrm{est}_i)$	Minimizing total delay
$\sum_{i,t} w_i(S_i)$	Minimizing costs per task per time
$\sum_i w(S_i)$	Minimizing costs per time

Table 2.3 – Objective possibilities depending on the value of γ.

For instance, $1|\,\mathrm{est}_j, \mathrm{pmtn}|E_{\max}$ denotes the scheduling problem on a single machine where the tasks

can be preempted. The release times of the tasks are given and the objective is to minimize the makespan.

2.2 Global Constraints Used in Scheduling

Constraint programming offers a powerful technique to model and solve scheduling problems. Actually, a scheduling problem can be recast as a CSP instance, by considering the set of starting times $X = \{S_1, ..., S_n\}$ to be the set of variables, where $\text{dom}(S_i) = [\text{est}_i, \text{lst}_i]$. Several constraint programming systems such as CHIP [1], ILOG Scheduler [59], Choco [60] and Gecode [31], have been developed which are equipped with packages designed for scheduling applications.

This section describes six prevalent global constraints for this framework. For the scheduling models to be the mentioned, the objective function γ is left undefined, as it is independent from the constraint.

2.2.1 ALL-DIFFERENT constraint

Definition 2.2.1. Let $X = \{X_1, ..., X_n\}$ be the set of variables. The constraint ALL-DIFFERENT$([X_1, \ldots, X_n])$ is satisfied if and only if $X_i \neq X_j$ for $1 \leq i \neq j \leq n$.

ALL-DIFFERENT constraint models $1|\,\text{est}_i, p_i = 1, \text{lct}_i\,|\gamma$. ALL-DIFFERENT$(S_1, ..., S_n)$ holds if the starting times take distinct values.

There exist several constraint propagation algorithms for the ALL-DIFFERENT constraint. For domain consistency of ALL-DIFFERENT, Régin [66] gives an $O(n^{5/2})$ algorithm. For the range consistency of ALL-DIFFERENT, Leconte [44] presents a quadratic algorithm based on identifying Hall intervals. Also, Quimper [62] develops an algorithm that runs in linear time when amortized over a branch of the search tree. For bounds consistency of ALL-DIFFERENT, Puget [61] gives an $O(n\log(n))$ algorithm. Mehlhorn and Thiel [52] present an algorithm for the bounds consistency that is $O(n)$ plus the time needed to sort the bounds of the domains. Furthermore, López-Ortiz, et al. [50] propose a linear algorithm with such a behaviour which turns out to be faster in practice. This algorithm has a smaller hidden multiplicative constant.

2.2.2 Global Cardinality

The *global cardinality constraint* (GCC) is a generalization of the ALL-DIFFERENT constraint. While the ALL-DIFFERENT constraint restricts each value to be assigned to at most one variable, the global cardinality constraint restricts each value to be assigned to a specified minimum and maximum number of variables.

Let $V = \{v_1, ..., v_l\}$ be a set of integers and let the interval $[a_i, b_i]$ be associated to each $v_i \in V$, $1 \leq i \leq l$. Let $X = \{X_1, ..., X_n\}$ be the set of variables and for $1 \leq i \leq n$, $\text{dom}(X_i) \subseteq V$. If

$occ(v_i, [X_1, \ldots, X_n])$ denotes the number of times a value $v_i \in V$ is assigned to a variable $X_j \in X$, $1 \leq j \leq n$, GCC($[X_1, \ldots, X_n], \vec{a}, \vec{b}$) ensures that $a_i \leq occ(v_i, [X_1, \ldots, X_n]) \leq b_i$.

GCC models $P|\,\mathrm{est}_i, p_i = 1, \mathrm{lct}_i\,|\gamma$. The model of GCC as a scheduling problem is similar to the one presented for ALL-DIFFERENT constraint. By setting $a_i = 0$ and $b_i = \mathcal{C}$, the constraint GCC($[S_1, \ldots, S_n], \vec{a}, \vec{b}$) models the problem where at most $\mathcal{C}$ tasks can start at any time t. Indeed, GCC can encode the ALL-DIFFERENT constraint if $a_i = 0$ and $b_i = 1$ for all i and all values in V.

Régin [67] first introduced this constraint and proposed an algorithm, achieving domain consistency in $O(n^2 l)$. Quimper, et.al [64] proposed a linear time algorithm for bounds consistency of this constraint.

2.2.3 INTER-DISTANCE

The INTER-DISTANCE(X, p) constraint ensures that the distance between any pair (X_i, X_j) of variables $[X_1, \ldots, X_n]$ is not smaller than a given value p. Formally, the INTER-DISTANCE holds if for all $i \neq j,\ \ |X_i - X_j| \geq p$. Note that when the minimum distance equals 1, the INTER-DISTANCE constraint reduces to the ALL-DIFFERENT constraint.

INTER-DISTANCE models $1|\,\mathrm{est}_i, p_i = p, \mathrm{lct}_i\,|\gamma$. The model of INTER-DISTANCE as a CSP is similar to the one introduced for ALL-DIFFERENT constraint, except that in any time window of size p, one might assign at most one value to a variables S_i.

This constraint was introduced by Régin [68]. Artiouchine and Baptiste [4] achieve bounds consistency in $O(n^3)$. Later on, Quimper et.al, [63] improved the complexity to $O(n^2)$. Enforcing domain consistency on this constraint is known to be NP-hard [6].

2.2.4 MULTI-INTER-DISTANCE

The MULTI-INTER-DISTANCE($[X_1, \ldots, X_n], m, p$) constraint is a generalization of INTER-DISTANCE which holds if for all v, $|\{i : X_i \in [v, v+p)\}| \leq m$. That is, at most m values taken from an interval of length p can be assigned to the variables X_i. Note that when $m = 1$, the MULTI-INTER-DISTANCE constraint specializes into an INTER-DISTANCE constraint. MULTI-INTER-DISTANCE models $P|\,\mathrm{est}_i, p_i = p, \mathrm{lct}_i\,|\gamma$. The model of MULTI-INTER-DISTANCE as a CSP is similar to the one introduced for ALL-DIFFERENT constraint, except that in any time window of size p, one might assign at most m values to the variables S_i. Ouellet and Quimper [57] introduced this constraint and proposed a propagator algorithm, achieving bounds consistency in cubic time.

2.2.5 Disjunctive constraint

Disjunctive scheduling is one of the most studied topics in scheduling. Filtering methods for DISJUNCTIVE constraint can be traced back prior to CP era [25, 43, 15]. This constraint prevents the tasks to overlap in time. Formally, the classical DISJUNCTIVE($[S_1, \ldots, S_n], \vec{p}$) constraint is satisfied if there is a precedence between arbitrary pairs of tasks. That is, for disjoint tasks $i \neq j \in \mathcal{I}$, it en-

forces the disjunctive condition $S_i + p_i \leq S_j$ or $S_j + p_j \leq S_i$. The DISJUNCTIVE constraint models $1|\operatorname{est}_i, p_i, \operatorname{lct}_i|\gamma$. This constraint can encode an INTER-DISTANCE constraint when $p_i = p$. However, encoding an INTER-DISTANCE constraint with a DISJUNCTIVE constraint hinders the filtering. It is NP-Hard to achieve bounds consistency on this constraint [5]. Chapter 3 presents filtering techniques for this constraint

Notice that DISJUNCTIVE constraint is also referred to as a *unary resource* constraint in CP literature.

2.2.6 CUMULATIVE constraint

The CUMULATIVE constraint, introduced by Aggoun and Beldiceanu [1], is the most general constraint which encodes all the global constraints introduced so far. The decision problem related to the CUMULATIVE constraint is called the *cumulative scheduling problem* (CuSP). It models $P|\operatorname{est}_i, p_i, \operatorname{lct}_i|\gamma$. This constraint models a relationship between a scarce resource $R \in \mathcal{R}$ and the tasks which are to be processed on R. Formally, the constraint CUMULATIVE($[S_1, \ldots, S_n], \vec{p}, \vec{c}, \mathcal{C}$) holds if and only if

$$\forall t : \sum_{S_i \leq t < S_i + p_i} c_i \leq \mathcal{C}$$

Admittedly, the CUMULATIVE constraint implies that the total rate of resource consumption by all the tasks underway at time t should not exceed $\mathcal{C}$.

This constraint encodes the DISJUNCTIVE constraint as well as the MULTI-INTER-DISTANCE constraint. Thus, by transitivity, it encodes all other constraints presented in the preceding sections. One can regard the MULTI-INTER-DISTANCE constraint as a CUMULATIVE constraint, where for all t, $\sum_{j|t \leq S_j < t+p} 1 \leq m$. That is, the total number of resources, simultaneously executing the tasks in a window of size p does not exceed m.

Determining whether CuSP has a solution is strongly NP-complete [10]. Thereby, it is NP-hard to enforce bounds consistency on this constraint. Notwithstanding, there exist several filtering rules running in polynomial time. Such rules will be expounded in chapter 3.

2.3 Polynomial time algorithms for scheduling problems

In spite of the NP-hardness of general scheduling problems with the three-field notation $\alpha|\beta|\gamma$, it turns out that this form is sensitive enough to describe scheduling problems for special cases for which polynomial time algorithms exist. Particularly, some scheduling problems can be reduced to well-known combinatorial optimization problems, such as maximum flow problems. This section surveys

$$P \mid \operatorname{est}_j; p_j = p; \operatorname{lct}_j \mid \gamma \tag{2.1}$$

For $\gamma \in \{E_{\max}, \sum_i E_i\}$, (2.1) falls into the category of scheduling problems for which polynomial time algorithms exist. There exists some objective functions that would make the problem NP-Hard, too.

2.3.1 Related Work

If $p = 1$ in (2.1), the problem is equivalent to finding a matching in a convex bipartite graph. Lipski and Preparata [47] present an algorithm running in $O(n\alpha(n))$ where α is the inverse of Ackermann's function. Gabow and Tarjan [28] reduce this complexity to $O(n)$ by using a restricted version of the Union-Find data structure.

Baptiste proves that in general, if γ can be expressed as the sum of n functions f_i of the completion time of each task i, where f_i's are non-decreasing and for any pair of jobs (i, j) the function $f_i - f_j$ is monotonous, the problem can be solved in polynomial time [8].

If the objective function γ is to minimize the maximum lateness $L_{\max}$, $P \mid \mathrm{est}_i, p_i = p \mid L_{\max}$ is polynomial [76] and $1 \mid \mathrm{est}_j, p_j = p \mid L_{\max}$ is solvable in $O(n \log n)$ [39].

Simons [74] presented an algorithm with the time complexity $O(n^3 \log\log(n))$ that solves (2.1). It is reported [46] that it minimizes both the sum of the completion times $\sum_j E_j$ as well as the makespan. Simons and Warmth [75] further improved the algorithm complexity to $O(mn^2)$. Dürr and Hurand [24] reduced the problem to a shortest path in a digraph and designed an algorithm in $O(n^4)$. This led López-Ortiz and Quimper [48] to introduce the idea of the scheduling graph.

2.3.2 Scheduling Graph

For the scheduling problems where all processing times are equal, López-Ortiz and Quimper [48] introduced the *scheduling graph* which conveys important properties. For instance, it allows to decide whether an instance is feasible, i.e. whether there exists at least one solution. The graph is based on the assumption that it is sufficient to determine how many tasks start at a given time. If one knows that there are h_t tasks starting at time t, it is possible to determine which tasks start at time t by computing a matching in a bipartite convex graph [48].

The scheduling problem (2.1) can be written as a CSP where the constraints are uniquely posted on the variables h_t. As a first constraint, we force the number of tasks starting at time t to be non-negative.

$$\forall\ \mathrm{est}_{\mathcal{I}} \leq t \leq o_{\mathcal{I}} - 1 \qquad\qquad h_t \geq 0 \tag{2.2}$$

At most m tasks can start within any window of size p.

$$\forall\ \mathrm{est}_{\mathcal{I}} \leq t \leq o_{\mathcal{I}} - p \qquad\qquad \sum_{j=t}^{t+p-1} h_j \leq m \tag{2.3}$$

At most n tasks can start within $[\mathrm{est}_{\mathcal{I}}, o_{\mathcal{I}})$.

$$\sum_{j=\mathrm{est}_{\mathcal{I}}}^{o_{\mathcal{I}}-1} h_j \leq n \tag{2.4}$$

Given two arbitrary (and possibly identical) tasks i and j, the set $K_{ij} = \{k : \text{est}_i \leq \text{est}_k \wedge o_k \leq o_j\}$ denotes the jobs that must start in the interval $[\text{est}_i, o_j)$. Thus,

$$\forall\, i,j \in \{1,\ldots,n\} \qquad \sum_{t=\text{est}_i}^{o_j-1} h_t \geq |K_{ij}| \tag{2.5}$$

In the context where the tasks have equal processing times, a solution that minimizes the sum of the completion times necessarily minimizes the sum of the starting time [48]. Thus, we consider these two objectives equivalent. The objective function of minimizing the sum of the starting times can also be written with the variables h_t.

$$\min \sum_{t=\text{est}_\mathcal{I}}^{o_\mathcal{I}-1} t \cdot h_t \tag{2.6}$$

To simplify the inequalites (2.2) to (2.5), we proceed to a change of variables. Let $x_t = \sum_{i=\text{est}_\mathcal{I}}^{t-1} h_i$, for $\text{est}_\mathcal{I} \leq t \leq o_\mathcal{I}$, be the number of tasks starting to execute before time t. Therefore, the constraints (2.2) to (2.5) can be rewritten as follows.

$$\forall\ \text{est}_\mathcal{I} \leq t \leq o_\mathcal{I} - 1 \qquad x_t - x_{t+1} \leq 0 \tag{2.7}$$

$$\forall\ \text{est}_\mathcal{I} \leq t \leq o_\mathcal{I} - p \qquad x_{t+p} - x_t \leq m \tag{2.8}$$

$$x_{o_\mathcal{I}} - x_{\text{est}_\mathcal{I}} \leq n \tag{2.9}$$

$$\forall\ \text{est}_i + 1 \leq o_j \qquad x_{\text{est}_i} - x_{o_j} \leq -|K_{ij}| \tag{2.10}$$

These inequalities form a system of difference constraints which can be solved by computing shortest paths in what is call the *scheduling graph* [48]. The scheduling graph G has for vertices the nodes $V(G) = \{\text{est}_\mathcal{I}, \ldots, o_\mathcal{I}\}$ and for edges $E(G) = E_f(G) \cup E_b(G) \cup E_n(G)$ where $E_f(G) = \{(t, t+p) : \text{est}_\mathcal{I} \leq t \leq o_\mathcal{I} - p\} \cup \{(\text{est}_\mathcal{I}, o_\mathcal{I})\}$ is the set of *forward edges* (from the inequalities (2.8) and (2.9)), $E_b(G) = \{(o_j, \text{est}_i) : \text{est}_i < o_j\}$ is the set of *backward edges* (from the inequality (2.10)), and $E_n(G) = \{(t+1, t) : \text{est}_\mathcal{I} \leq t < o_\mathcal{I}\}$ is the set of *null edges* (from the inequality (2.7)). The following weight function maps every edge $(a, b) \in E(G)$ to a weight:

$$w'(a,b) = \begin{cases} m & \text{if } a + p = b \\ n & \text{if } a = \text{est}_\mathcal{I} \wedge b = o_\mathcal{I} \\ -|\{k : b \leq \text{est}_k \wedge o_k \leq a\}| & \text{if } a > b \end{cases} \tag{2.11}$$

Figure 2.3 illustrates the scheduling graph of five tasks whose processing times is $p = 2$.

Theorem 1 shows how to compute a feasible schedule from the scheduling graph.

Theorem 1 (López-Ortiz and Quimper [48])**.** Let $\delta(a, b)$ be the shortest distance between node a and node b in the scheduling graph. The assignment $x_t = n + \delta(o_\mathcal{I}, t)$ provides a solution to the problem 2.1 subject to the constraints (2.8) to (2.10) that minimizes the sum of the completion times ($\gamma = \sum_i E_i$).

i	est_i	o_i
1	4	8
2	1	4
3	1	6
4	1	9
5	1	6

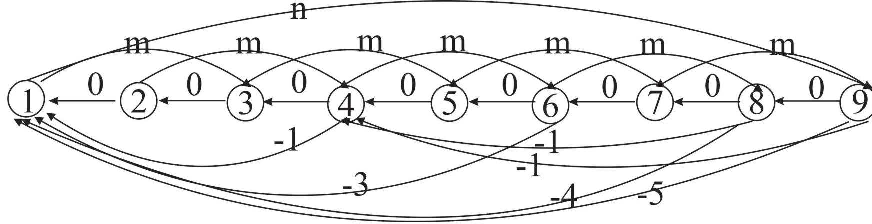


Figure 2.3 – The scheduling graph with five tasks with processing times $p = 2$.

The scheduling problem has a solution if and only if the scheduling graph has no negative cycles. An adaptation of the Bellman-Ford algorithm finds a schedule by computing the shortest path in this graph [48]. The algorithm runs in $O(\min(1, \frac{p}{m})n^2)$, which is sub-quadratic when $p < m$ and quadratic otherwise. The schedule minimizes both $\sum_j E_j$ and $E_{\max}$.

2.3.3 Network Flows

Definition 2.3.1. Let $\overrightarrow{N}$ be a digraph with the vertices $V(\overrightarrow{N})$ and edges $E(\overrightarrow{N})$ where each edge $(i, j) \in E(\overrightarrow{N})$ has a flow capacity u_{ij} and a flow cost w_{ij}. There is one node $s \in V(\overrightarrow{N})$ called the *source* and one node $t \in V(\overrightarrow{N})$ called the *sink*. A *flow* is a vector that maps each edge $(i, j) \in E(\overrightarrow{N})$ to a value x_{ij} such that the following constraints are satisfied.

$$0 \leq x_{ij} \leq u_{ij} \tag{2.12}$$

$$\sum_{j \in V(\overrightarrow{N})} x_{ji} - \sum_{j \in V(\overrightarrow{N})} x_{ij} = 0 \qquad \forall i \in V(\overrightarrow{N}) \setminus \{s, t\} \tag{2.13}$$

The *residual network* with respect to a given flow x is formed with the same nodes $V(\overrightarrow{N})$ as the original network. However, for each edge (i, j) such that $x_{ij} < u_{ij}$, there is an edge (i, j) in the residual network of cost w_{ij} and residual capacity $u_{ij} - x_{ij}$. For each edge (i, j) such that $x_{ij} > 0$, there is an edge (j, i) in the residual network of cost $-w_{ij}$ and residual capacity x_{ij}. Figure 2.4 illustrates a network flow along with its residual network, where the numbers assigned to every edge (i, j) in the network flow are in the form of x_{ij}/u_{ij} and in the residual network they refer to x_{ij}.

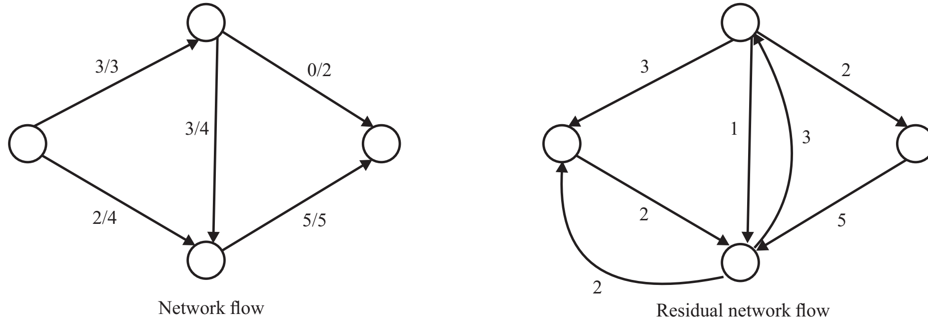


Figure 2.4 – A network flow and a residual network flow.

The *min-cost flow* satisfies the constraints (2.12) and (2.13) while minimizing $\sum_{(i,j)\in E(\overrightarrow{N})} w_{ij}x_{ij}$.

Definition 2.3.2. A matrix with entries in $\{-1, 0, 1\}$ which has precisely one 1 and one -1 per column is called a *network matrix*.

If A is a network matrix, the following optimization model formulates a flow.

$$\text{Maximize } w^T x, \text{ subject to } \begin{cases} Ax = b \\ x \geq 0 \end{cases} \tag{2.14}$$

Vice versa, the linear program can also be encoded as a flow.

There is one node for each row of the matrix in addition to a source node s and a sink node t. Each column in the matrix corresponds to an edge $(i, j) \in E(\overrightarrow{N})$ where i is the node whose row is set to 1 and j is the node whose column is set to -1. If $b_i > 0$ we add the edge (i, t) of capacity b_i and if $b_i < 0$ we add the edge (s, i) of capacity $-b_i$ [92].

To our knowledge, the *successive shortest path algorithm* is the state of the art, for this particular structure of the network, to solve the min-cost flow problem. This algorithm successively augments the flow values of the edges along the shortest path connecting the source to the sink in the residual graph. Let $W = \max_{(i,j)\in E(\overrightarrow{N})} |w_{ij}|$ be the greatest absolute cost and $B = \max_{i\in V(\overrightarrow{N})} b_i$ be the largest value in the vector b. To compute the shortest path, one can use Goldberg's algorithm [32] with a time complexity of $O(|E(\overrightarrow{N})|\sqrt{|V(\overrightarrow{N})|}\log(W))$. Since at most $|V(\overrightarrow{N})|B$ shortest path computations are required, this leads to a time complexity of $O(|V(\overrightarrow{N})|^{1.5}|E(\overrightarrow{N})|\log(W)B)$.

Chapter 3

Filtering algorithms for the disjunctive and cumulative constraints

Most of the scheduling problems turn out to be NP-hard [29]. Particularly, enforcing bounds consistency for the CUMULATIVE constraint is NP-Hard. Due to that, several sophisticated filtering algorithms such as Time-Tabling, Detectable Precedences, Edge-Finding, Not-First/Not-Last and Energetic Reasoning which reason on a relaxation of the problem, were introduced in the last two decades. These algorithms run in polynomial time. This section presents the state of the art on such filtering algorithms to the CUMULATIVE and DISJUNCTIVE constraints, in order to quickly detect whether the constraints are violated, not to mention to prune the search space.

Roughly, the *compulsory part* of a task is a time interval in which the execution of the task is inevitable. For the algorithms to be presented in this chapter, Time-Tabling filters the CUMULATIVE constraint by exploiting the compulsory parts of the tasks. The alternative filtering algorithms discuss some classical propagation techniques which reason about the order of the execution of the tasks. Technically, these techniques rely on the analysis of the interaction between the tasks, by aiming to discover whether a task must execute before or after a given group of tasks. Notably, the filtering techniques proceed by taking into account the position of a task with respect to the subsets of tasks executing on the same resource. Detecting such ordering is important to eliminate values which do not contribute in a feasible solution. Another aspect of this is that some of these rules provide a stronger filtering than others and some rules require more time to enforce than others. Thus, there is a trade-off between the amount of filtering and the time that is elapsed during the filtering.

Prior to presenting the algorithms, we establish the following convention. Let A be a task and Θ be a set of tasks. For the CUMULATIVE constraint, if A executes after Θ, in all solutions A completes after Θ. This case is denoted $\Theta \prec A$. For the DISJUNCTIVE constraint if A executes after Θ , in all solutions A starts after Θ terminates. This case is denoted $\Theta \ll A$.

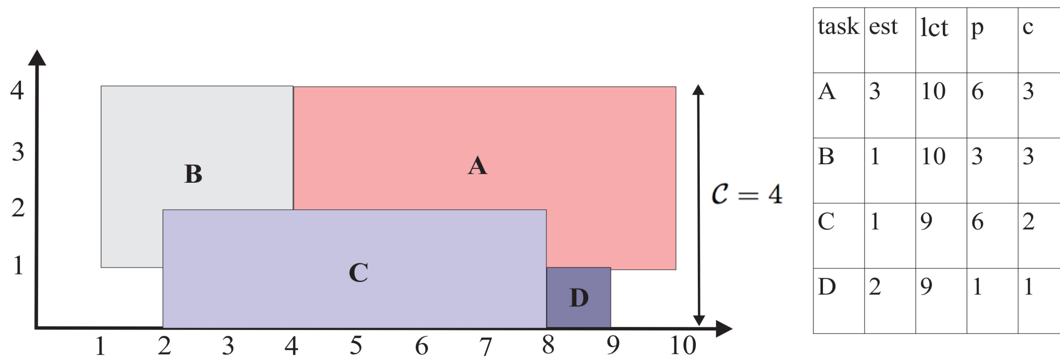
3.1 Overload Checking

Let $\Theta \subseteq \mathcal{I}$ be a subset of tasks subject to the CUMULATIVE constraint. The *Overload Checking* rule ensures that the workload of Θ does not exceed the total available energy inside the window where the tasks of Θ execute. That is,

$$\mathrm{e}_\Theta \leq \mathcal{C}(\mathrm{lct}_\Theta - \mathrm{est}_\Theta) \tag{3.1}$$

If (3.1) holds for all $\Theta \subseteq \mathcal{I}$, the problem is said to be *e-feasible*.

Example 3.1.1. If $\Theta = \{A, B, C, D\}$ for the instance of figure 3.1, the Overload Checking triggers a failure. In fact, there is not enough energy in the whole window where the tasks of Θ can execute, for $\mathrm{e}_\Theta = 40 = 18 + 9 + 12 + 1 > 4(10 - 1) = 36$.



task	est	lct	p	c
A	3	10	6	3
B	1	10	3	3
C	1	9	6	2
D	2	9	1	1

Figure 3.1 – Overload Checking triggers a failure for $\Theta = \{A, B, C, D\}$.

Evidently, passing the Overload Checking provides a necessary condition for the existence of a solution. In contrast, there are instances for which there is not a feasible solution, whereas the Overload Checking does not fail.

Example 3.1.2. Consider a set of tasks with the information in table 3.1 and let $\mathcal{C} = 4$. The Overload Checking does not detect a failure for this instance, while the instance is not feasible.

task	est	lct	p	c
A	14	23	9	4
B	0	6	6	3
C	6	28	8	4
D	6	24	4	4

Table 3.1 – A set of tasks for which neither Overload Checking fails, nor is there a feasible solution.

Armin Wolf and Gunnar Schrader [91] introduced an algorithm for Overload Checking which runs in $O(n\log(n))$. Vilím [89] designed a sophisticated data structure in the form of a balanced binary tree, to present an algorithm which achieves the same computational effort.

Checking all subsets $\Theta \subseteq \mathcal{I}$ that satisfy (3.1) can be too time consuming. In the following we introduce a more efficient way for this purpose.

Definition 3.1.1. [89] Let $\Theta \subseteq \mathcal{I}$. The *earliest energy envelope* and the *latest energy envelope* of Θ are respectively defined by

$$\mathrm{Env}_\Theta = \max_{\Omega \subseteq \Theta} (\mathcal{C}\, \mathrm{est}_\Omega + \mathrm{e}_\Omega) \tag{3.2}$$

$$\mathrm{Env}'_\Theta = \min_{\Omega \subseteq \Theta} (\mathcal{C}\, \mathrm{lct}_\Omega - \mathrm{e}_\Omega) \tag{3.3}$$

Env_Θ provides the aggregation of the energy which is required to fully use the resource up to the time est_Ω and the energy of executing the tasks in Ω. Note that the maximum in (3.2) can be obtained from a subset $\Omega \subset \Theta$, if $\mathrm{est}_\Omega > \mathrm{est}_\Theta$. A symmetric reasoning for Env'_Θ holds.

The following lemma, proposed by Vilím [89], provides an efficient way to assess (3.1) with taking advantage of the concept of earliest energy envelope.

Lemma 1. Let $\mathrm{Lcut}(\mathcal{I}, j) = \{l \in \mathcal{I} : \mathrm{lct}_l \leq \mathrm{lct}_j\}$. The problem is *e-feasible* if and only if $\forall j \in \mathcal{I}$: $\mathrm{Env}_{\mathrm{Lcut}(\mathcal{I},j)} \leq \mathcal{C} \cdot \mathrm{lct}_j$

In the following we introduce a data structure which is used to compute (3.2).

Definition 3.1.2. Let $\Theta \subseteq \mathcal{I}$ be a subset of tasks which are scheduled on the time line. The *cumulative* Θ_L-*tree* (or $\Theta-$tree, as it is called by Vilím [89]) is an essentially complete binary tree whose leaves consist of the tasks Θ inserted in ascending order of their release time from left to right. Each leaf node v of the tree includes the data related to the energy and the earliest energy envelope defined as below.

$$\mathrm{e}_v = \begin{cases} c_v p_v & \text{if } v \in \Theta \\ 0 & \text{otherwise} \end{cases} \tag{3.4}$$

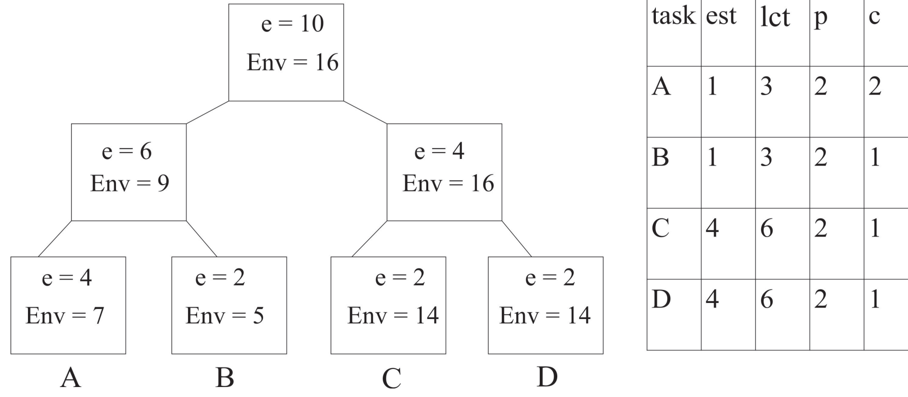
$$\mathrm{Env}_v = \begin{cases} \mathcal{C}\, \mathrm{est}_v + \mathrm{e}_v & \text{if } v \in \Theta \\ -\infty & \text{otherwise} \end{cases} \tag{3.5}$$

Each internal node v' of the tree, whose left and right children are respectively denoted left(v') and right(v'), includes the data related to the energy and the earliest energy envelope of those tasks from Θ which belong to the subtree rooted at v'. These values are recursively computed by

$$\mathrm{e}_{v'} = \mathrm{e}_{\mathrm{left}(v')} + \mathrm{e}_{\mathrm{right}(v')} \tag{3.6}$$

$$\mathrm{Env}_{v'} = \max\{\mathrm{Env}_{\mathrm{left}(v')} + \mathrm{e}_{\mathrm{right}(v')}, \mathrm{Env}_{\mathrm{right}(v')}\} \tag{3.7}$$

In particular, for the root of the Θ_L-tree, $\mathrm{Env}_{\mathrm{root}} = \mathrm{Env}_\Theta$.



task	est	lct	p	c
A	1	3	2	2
B	1	3	2	1
C	4	6	2	1
D	4	6	2	1

Figure 3.2 – The cumulative Θ_L-tree corresponding to $\mathcal{I} = \{A, B, C, D\}$ with a resource of capacity $\mathcal{C} = 3$.

Initializing the leaves in the cumulative Θ_L-tree requires $O(n)$ time. The genius of this tree is its ability to reuse previous computations by managing efficiently the set of tasks once an insertion or removal occurs. In fact, the update of Θ, when a new task is added or removed from Θ, merely affects the values of the inner nodes, lying on a path from the leaf to the root. Taking advantage of the structure of such a tree, all the parameters in the nodes of the tree can be computed in $O(n)$ and updated in $O(\log(n))$. Particularly, inserting a task to or removing a task from Θ runs in $O(\log(n))$.

Example 3.1.3. Figure 3.2 illustrates the cumulative Θ_L-tree corresponding to a set of tasks $\mathcal{I} = \{A, B, C, D\}$ and with $\Theta = \mathcal{I}$ as represented, with a resource of capacity $\mathcal{C} = 3$.

Vilím [89] takes advantage of the cumulative Θ_L-tree to propose the algorithm 1 for Overload Checking. This algorithms initializes an empty Θ_L-tree, i.e. $\Theta = \emptyset$, and inserts the tasks in ascending order of latest completion times in the tree. Inserting a task in the tree is equivalent to scheduling the task. After updating the tree, the earliest energy envelope of the set of tasks which are scheduled so far is retrieved from the root of the tree and e-feasibility is checked to detect inconsistencies according to lemma 1. The algorithm admits a running time of $O(n \log(n))$.

Algorithm 1: `OverloadChecking`($\mathcal{I}$)

1 $\Theta_L \leftarrow \emptyset$;
2 **for** $j \in \mathcal{I}$ *in non-decreasing order of* lct_j **do**
3 $\quad\Theta_L \leftarrow \Theta_L \cup \{j\}$;
4 $\quad$**if** $\text{Env}_{\Theta_L} > \mathcal{C} \cdot \text{lct}_j$ **then**
5 $\quad\quad$**fail;**

3.2 Time-Tabling

Time-Tabling is a filtering technique to filter the CUMULATIVE constraint. This constraint maintains a minimal resource consumption at each time t, which allows the solver to restrict the domains of other tasks by preventing them from executing at times that would lead to the over-consumption of the resource.

Definition 3.2.1. Let $i \in \mathcal{I}$. If $\text{lst}_i < \text{ect}_i$, the time window $[\text{lst}_i, \text{ect}_i)$ is called the *compulsory part* of i. That is, if it exists, a compulsory part is a time window in which a task certainly executes.

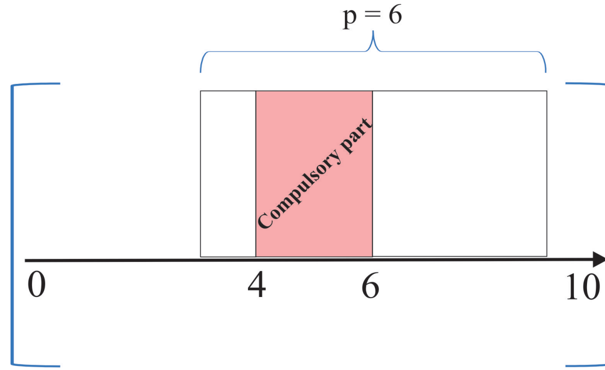


Figure 3.3 – The compulsory part of the task A with $\text{est}_A = 0, \text{lct}_A = 10, p_A = 6$ is [4,6).

Figure 3.3 represents a task which has a compulsory part.

A compulsory part allows a reservation of a resource amount and prevents the other tasks to have access to that resource amount. Let $f(t, \Theta)$ denote the amount of energy that is consumed by the tasks of Θ for which t lies in their compulsory parts:

$$f(t, \Theta) = \sum_{i \in \Theta | \text{lst}_i \leq t < \text{ect}_i} c_i$$

In other words, $f(t, \Theta)$ provides a lower bound on the resource consumption at time t. The following formulae capture the Time-Tabling rules for filtering earliest starting times and latest completion times. The left side of the implication corresponds to the detection test and the right side corresponds to the adjustment.

$$(c_i + f(t, \mathcal{I} \setminus \{i\}) > \mathcal{C}) \wedge (t < \text{ect}_i) \Rightarrow \text{est}_i > t \quad (3.8)$$

$$(c_i + f(t, \mathcal{I} \setminus \{i\}) > \mathcal{C}) \wedge (\text{lst}_i \leq t) \Rightarrow \text{lct}_i \leq t \quad (3.9)$$

Notice that after applying the rules, the same task might get filtered further with respect to another time t.

Several algorithms apply the Time-Tabling rules. Beldiceanu et al. [12] propose the sweep algorithm which iterates through the time by gradually enlarging the aggregation of compulsory parts and applying the pruning. This method is improved in [45] which copes with massive sets of tasks. Ouellet et

al. [56] present an algorithm in $O(n\log(n))$, which is motivated from a task decomposition technique. As a contribution, in chapter 4 we will propose a new algorithm for Time-Tabling in linear time for the DISJUNCTIVE constraint [26]. Recently, Gay et. al. [30] were inspired from our linear-time algorithm for Time-Tabling and they also proposed a linear-time Time-Tabling for the CUMULATIVE constraint. However, they present another algorithm which has a quadratic worst-case complexity that nevertheless performs better in practice.

3.3 Edge-Finding

Edge-Finding is another filtering technique for the CUMULATIVE constraint that complements the Time-Tabling. It aims at finding relations of precedences between the tasks. If G is a graph whose nodes represent the tasks and its edges indicate the possible orderings between tasks, Edge-Finding "in a sense" creates new edges in G by detecting precedences between the tasks. In fact, detecting precedences creates new ordering relations, which are interpreted as the edges of G. Furthermore, the Edge-Finding synchronizes the temporal time bounds of tasks by filtering the values which do not yield a solution. Formally, in the Edge-Finding technique, a subset $\Theta \subseteq \mathcal{I}$ is assumed and a task $i \in \mathcal{I} \setminus \Theta$ is constrained to execute first (or last) with respect to Θ. The following rules capture the bounding techniques of Edge-Finding:

$$\forall \Theta, \forall i \notin \Theta, C(\mathrm{lct}_\Theta - \mathrm{est}_{\Theta \cup \{i\}}) < \mathrm{e}_{\Theta \cup \{i\}} \Rightarrow \Theta \prec i \tag{3.10}$$

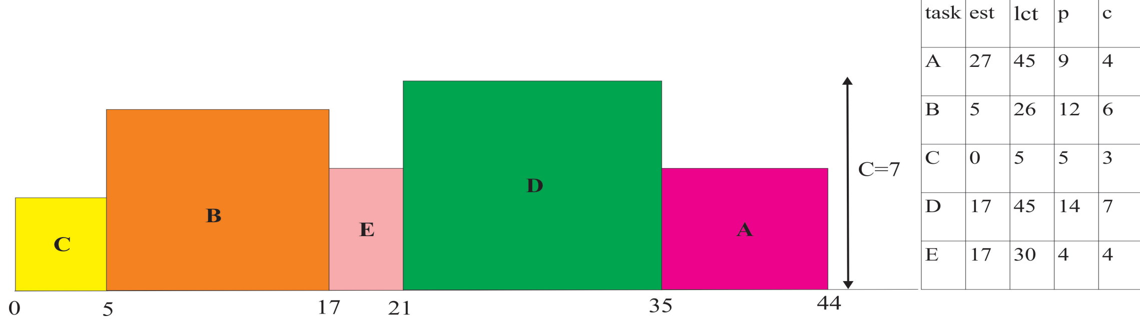
$$\forall \Theta, \forall i \notin \Theta, C(\mathrm{lct}_{\Theta \cup \{i\}} - \mathrm{est}_\Theta) < \mathrm{e}_{\Theta \cup \{i\}} \Rightarrow i \prec \Theta \tag{3.11}$$

The idea for (3.10) can be deduced from the Overload Checking, in the sense that if the scheduling of task i at its release time causes an overload in the entire window where Θ can execute, then Θ has to precede i. A symmetric reasoning for (3.11) in the reverse direction holds.

Once a precedence $\Theta \prec i$ or $i \prec \Theta$ is detected, one can adjust the earliest starting time or the latest completion time of i. For $\Omega \subseteq \Theta$, there are e_Ω units of energy within the entire window where Ω can execute. Let

$$\mathrm{rest}(\Omega, c_i) = \mathrm{e}_\Omega - (\mathcal{C} - c_i)(\mathrm{lct}_\Omega - \mathrm{est}_\Omega) \tag{3.12}$$

It is proven that the subsets which have enough energy to prevent concurrent scheduling of i are those for which $\mathrm{rest}(\Omega, c_i) > 0$ and if so, $\mathrm{est}_\Omega + \left\lceil \frac{\mathrm{rest}(\Omega, c_i)}{c_i} \right\rceil$ provides a lower bound for est_i. Likewise, $\mathrm{lct}_\Omega - \left\lceil \frac{\mathrm{rest}(\Omega, c_i)}{c_i} \right\rceil$ provides an upper bound for lct_i [11]. The following relations for the new bounds are deduced:



task	est	lct	p	c
A	27	45	9	4
B	5	26	12	6
C	0	5	5	3
D	17	45	14	7
E	17	30	4	4

Figure 3.4 – The precedence $\Theta_1 = \{B, E\} \prec D$ gives rise to filtering $\text{est}_D = 20$ and the precedences $D \prec \Theta_2 = \{A\}$ and $B \prec \Theta_3 = \{A, D, E\}$ give rise to $\text{lct}_D = 40, \text{lct}_B = 25$.

$$\text{est}_i \leftarrow \max(\text{est}_i, \max_{\substack{\emptyset \neq \Omega \subseteq \Theta \\ \text{e}_\Omega > (\mathcal{C} - c)(\text{lct}_\Omega - \text{est}_\Omega)}} \{\text{est}_\Omega + \left\lceil \frac{\text{rest}(\Omega, c_i)}{c_i} \right\rceil\}) \tag{3.13}$$

$$\text{lct}_i \leftarrow \min(\text{lct}_i, \min_{\substack{\emptyset \neq \Omega \subseteq \Theta \\ \text{e}_\Omega > (\mathcal{C} - c)(\text{lct}_\Omega - \text{est}_\Omega)}} \{\text{lct}_\Omega - \left\lceil \frac{\text{rest}(\Omega, c_i)}{c_i} \right\rceil\}) \tag{3.14}$$

The maximum (minimum) should be taken over all subsets $\emptyset \neq \Omega \subseteq \Theta$, because a proper subset Ω can give rise to a stronger filtering. Besides, $\text{e}_\Omega > (\mathcal{C} - c)(\text{lct}_\Omega - \text{est}_\Omega)$ is checked to ensure that Ω has sufficient energy to be in potential overlap with i.

Example 3.3.1. Consider the set of tasks in figure 3.4 and let the capacity of the resource be $\mathcal{C} = 7$. For $\Theta_1 = \{B, E\}$ and $i = D$, the detection rule (3.10) indicates that $\Theta_1 \prec D$, since 72 + 16 + 98 = 186 > 7 (30 - 5) = 175. Therefore, one can increase the value of est_D using (3.13). The maximum is achieved for $\Omega = \{E\} \subset \Theta_1$ and since $\text{rest}(\{E\}, 7) = 16$, therefore est_D gets filtered to $\text{est}_D = 17 + \lceil 16/7 \rceil = 20$. Furthermore, the detection rule (3.11) implies that $D \prec \Theta_2 = \{A\}$, since 98 + 36 = 134 > 7(45 - 27) = 126. Therefore, the latest completion time of D can be filtered using (3.14), as well. In this case, $\Omega = \{A\}$ and since $\text{rest}(\Omega, 7) = 36$, therefore lct_D gets filtered to $\text{lct}_B = 45 - \lfloor 36/7 \rfloor = 40$. Moreover, the precedence $B \prec \Theta_3 = \{A, D, E\}$ holds, since 222 = 36 + 98 + 16 + 72 > 7(45 - 17) = 196. Therefore, one can decrease the value of lct_B using (3.14). The minimum is achieved for $\Omega = \Theta_3$ and since $\text{rest}(\Omega, 6) = (36 + 98 + 16) - (7 - 6)(45 - 17) = 122$, therefore lct_B gets filtered to $\text{lct}_B = 45 - \lfloor 122/6 \rfloor = 25$.

Mercier and Van Hentenryck [53] proved that the original Edge-Finding algorithm presented in [55] is incomplete and they presented an algorithm in $O(kn^2)$, where k denotes the number of distinct capacities associated to the tasks in $\mathcal{I}$. Later on, Vilím [84] improved the complexity to $O(kn \log(n))$. Kameugne et.al, present an Edge-Finding algorithm in $O(n^2)$. They empirically prove that their algorithm is substantially faster than Vilím's Edge-Finding [42]. Vilím's algorithm is based on the concept of a $(\Theta - \Lambda)_L$−tree that we present in the next section.

3.3.1 $(\Theta - \Lambda)_L-$tree

Note that (3.10) is equivalent to

$$\forall \Theta, \forall i \notin \Theta, \mathrm{Env}(\Theta \cup \{i\}) > \mathcal{C} \cdot \mathrm{lct}_\Theta \Rightarrow \Theta \prec i \tag{3.15}$$

Vilím introduced a new data structure based on the $\Theta-$tree that, in addition to computing the envelope of $\Theta \cup \{i\}$, helps finding out which task in Λ is the task i. The *cumulative* $(\Theta - \Lambda)_L-$*tree* introduced by Vilím [84], is an extension of the cumulative Θ_L-tree which allows to apply the detection and adjustment rules of the Edge-Finding. In this tree, $\Lambda \subseteq \mathcal{I} \setminus \Theta$ and there is a leaf for all task in $\mathcal{I}$, where each task can belong to Θ or Λ. Initially, all the tasks are in Θ and as the algorithm processes, the tasks are moved from Θ to Λ and later removed from Λ. Throughout its execution, the algorithm finds precedences of the form $\Theta \prec i$ and $i \in \Lambda$. In addition to the values e_v and Env_v for a leaf v, two more values e_v^Λ and Env_v^Λ are defined and they are computed as follows:

$$\mathrm{e}_v = \begin{cases} \mathrm{e}_i & \text{if } i \in \Theta \\ 0 & \text{if } i \in \Lambda \end{cases}, \mathrm{e}_v^\Lambda = \begin{cases} -\infty & \text{if } i \in \Theta \\ \mathrm{e}_i & \text{if } i \in \Lambda \end{cases}$$

$$\mathrm{Env}_v = \begin{cases} \mathcal{C}\,\mathrm{est}_i + \mathrm{e}_i & \text{if } i \in \Theta \\ -\infty & \text{if } i \in \Lambda \end{cases}, \mathrm{Env}_v^\Lambda = \begin{cases} -\infty & \text{if } i \in \Theta \\ \mathcal{C}\,\mathrm{est}_i + \mathrm{e}_i & \text{if } i \in \Lambda \end{cases}$$

e_v and Env_v are respectively the total energy and earliest energy envelope of all tasks in Θ whose associated leaf is a descendant of v. In particular, for the root of the tree, $\mathrm{e}_{\text{root}} = \mathrm{e}_\Theta$ and $\mathrm{Env}_{\text{root}} = \mathrm{Env}_\Theta$.

e_v^Λ and Env_v^Λ are the maximum energy and the earliest energy envelope of the tasks in Θ whose leaves are descendant of v to which one task from Λ is added. The task from Λ is also a descendant of v. In other words, $\mathrm{e}_v^\Lambda = \max_{i \in \Lambda} \mathrm{e}_{\Theta \cup \{i\}}$ and $\mathrm{Env}_v^\Lambda = \max_{i \in \Lambda} \mathrm{Env}_{\Theta \cup \{i\}}$.

Assuming

$$\mathrm{e}_{v'} = \mathrm{e}_{\text{Leaves}(v') \cap \Theta}$$

$$\mathrm{e}_{v'}^\Lambda = \mathrm{e}_{\text{Leaves}(v') \cap \Theta} + \max_{i \in \text{Leaves}(v') \cap \Lambda} \{\mathrm{e}_i\}$$

$$\mathrm{Env}_{v'} = \mathrm{Env}_{\text{Leaves}(v') \cap \Theta}$$

$$\mathrm{Env}_{v'}^\Lambda = \mathrm{Env}_{(\text{Leaves}(v') \cap \Theta, \text{Leaves}(v') \cap \Lambda)}$$

for an internal node v', they are computed as follows

$$\mathrm{e}_{v'} = \mathrm{e}_{\text{left}(v')} + \mathrm{e}_{\text{right}(v')}$$

$$\mathrm{e}_{v'}^\Lambda = \max\{\mathrm{e}_{\text{left}(v')}^\Lambda + \mathrm{e}_{\text{right}(v')}, \mathrm{e}_{\text{left}(v')} + \mathrm{e}_{\text{right}(v')}^\Lambda\}$$

$$\mathrm{Env}_{v'} = \max\{\mathrm{Env}_{\text{left}(v')} + \mathrm{e}_{\text{right}(v')}, \mathrm{Env}_{\text{right}(v')}\}$$

$$\mathrm{Env}_{v'}^\Lambda = \max\{\mathrm{Env}_{\text{left}(v')}^\Lambda + \mathrm{e}_{\text{right}(v')}, \mathrm{Env}_{\text{left}(v')} + \mathrm{e}_{\text{right}(v')}^\Lambda, \mathrm{Env}_{\text{right}(v')}^\Lambda\}$$

The $(\Theta - \Lambda)_L-$tree extends the Θ_L-tree and provides the possibility to compute the value of the earliest energy envelope that includes all the tasks in Θ and at most one task from Λ. Particularly, $(\Theta - \Lambda)_L-$tree is used to quickly locate the tasks that could be involved in (3.15). This way leads to find the strongest ordering for each task with respect to a set. The algorithm that Vilím proposes for Edge-Finding runs in two steps: the detection algorithm, which uses $(\Theta - \Lambda)_L-$tree to detect the orderings, and the adjustment of time bounds. This phase also uses the Θ_L-tree to compute the subsets of task involved in (3.13) which contribute to the strongest adjustment. Thanks to the structure of $(\Theta-\Lambda)_L-$tree and Θ_L-tree, the entire algorithm runs in $O(kn\log(n))$. Further, Scott [73] presents a detailed implementation of Vilím's algorithm.

3.4 Extended-Edge-Finding

Despite the power of the Edge Finding in pruning the search space, the bottleneck of Edge-Finding is that it does not heed the tasks that overlap a set Θ. The Extended-Edge-Finding is an extension of the Edge-Finding which compensates for such a deficiency. The pruning rules of this technique for a set of activities $\Theta \subseteq \mathcal{I}$ and $i \notin \Theta$ are as follows

$$\mathrm{est}_i \leq \mathrm{est}_\Theta < \mathrm{ect}_i, \mathrm{e}_\Theta + c_i(\mathrm{ect}_i - \mathrm{est}_\Theta) > \mathcal{C}(\mathrm{lct}_\Theta - \mathrm{est}_\Theta) \Rightarrow \Theta \prec i \tag{3.16}$$

$$\mathrm{lst}_i < \mathrm{lct}_\Theta \leq \mathrm{lct}_i, \mathrm{e}_\Theta + c_i(\mathrm{lct}_\Theta - \mathrm{lst}_i) > \mathcal{C}(\mathrm{lct}_\Theta - \mathrm{est}_\Theta) \Rightarrow i \prec \Theta \tag{3.17}$$

The reasoning behind (3.16) for the Extended-Edge-Finding is the same reasoning as the Edge-Finding, except that when $\mathrm{est}_i < \mathrm{est}_\Theta$ at most $c_i(\mathrm{est}_\Theta - \mathrm{est}_i)$ units of energy of task i is spent outside $[\mathrm{est}_\Theta, \mathrm{lct}_\Theta]$. A similar reasoning in the reverse direction for (3.17) holds.

The following example shows that there might occur situations in which no deduction can be made by using the filtering rules of Edge-Finding, however it is still possible to derive a precedence with the Extended-Edge-Finding.

Example 3.4.1. Let $\mathcal{I} = \{A, B, C, D\}$ be a set of tasks whose parameters are given in Table 3.2.

task	est	lct	p	c
A	3	10	4	2
B	2	22	7	1
C	3	10	2	3
D	3	10	1	3

Table 3.2 – The information of a set of tasks $\mathcal{I} = \{A, B, C, D\}$ with a resource of capacity $\mathcal{C} = 3$.

The capacity of the resource equals $\mathcal{C} = 3$. Assuming $\Theta = \{A, C, D\}$, $\mathrm{e}_{\Theta\cup\{B\}} = \mathcal{C}(\mathrm{lct}_\Theta - \mathrm{est}_{\Theta\cup B})$. Hence, (3.10) deducts nothing. However, the Extended-Edge-Finding rule (3.16) detects $\Theta \prec B$.

Once the appropriate precedences are detected, the bounds are adjusted as

$$\mathrm{est}_i \leftarrow \max(\max_{\substack{\Theta \subseteq \mathcal{I} \\ i \notin \Theta \\ \beta(\Theta,i) \vee \alpha(\Theta,i)}} \max_{\substack{\Omega \subseteq \Theta \\ \mathrm{rest}(\Omega,c_i)>0}} \mathrm{est}_\Omega + \lceil \frac{\mathrm{rest}(\Omega, c_i)}{c_i} \rceil, \mathrm{est}_i) \tag{3.18}$$

$$\mathrm{lct}_i \leftarrow \min(\min_{\substack{\Theta \subseteq \mathcal{I} \\ i \notin \Theta \\ \beta(\Theta,i) \vee \alpha(\Theta,i)}} \min_{\substack{\Omega \subseteq \Theta \\ \mathrm{rest}(\Omega,c_i)>0}} \mathrm{lct}_\Omega - \lceil \frac{\mathrm{rest}(\Omega, c_i)}{c_i} \rceil, \mathrm{lct}_i) \tag{3.19}$$

where

$$\alpha(\Theta, i) \iff (\mathcal{C}(\mathrm{lct}_\Theta - \mathrm{est}_{\Theta \cup \{i\}}) < \mathrm{e}_\Theta + \mathrm{e}_i) \vee (\mathrm{ect}_i \geq \mathrm{lct}_\Theta) \tag{3.20}$$

$$\beta(\Theta, i) \iff (\mathcal{C}(\mathrm{lct}_\Theta - \mathrm{est}_\Theta) < \mathrm{e}_\Theta + c_i(\mathrm{est}_i + p_i - \mathrm{est}_\Theta)) \wedge (\mathrm{est}_i \leq \mathrm{est}_\Theta < \mathrm{ect}_i) \tag{3.21}$$

The right side of (3.18) is taken over all subsets Θ which make a precedence with i according to the rule (3.16). Since i cannot be executed on all of such areas, (3.18) is derived. A symmetric reasoning for (3.19) holds.

Nuijten [55] proposes an algorithm for the Extended-Edge-Finding filtering, which runs in $O(kn^3)$, where k denotes the number of distinct capacities. As mentioned in the last section, since Mercier and Van Hentenryck [53] proved that the Edge-Finding algorithm given in [55] is incomplete, so is the Extended-Edge-Finding proposed by Nuijten. Rather, they proposed an algorithm for the Extended-Edge-Finding in two phases, which runs in $O(kn^2)$. Kameugne et.al, [41] proposed a sound algorithm for the Extended-Edge-Finding running in $O(n^2)$. Ouellet et. al, [58] developed an improved algorithm with a time complexity of $O(kn \log(n))$.

3.5 Not-First/Not-Last

While the Edge finding and Extended-Edge finding techniques ensure if a task i must execute first or last in $\Theta \cup \{i\}$, Not-First/Not-Last is a complementary type of the Edge-Finding rule which determines whether i must execute either after at least one task or before at least one task of a set Θ. The filtering rules are as follows

$$\mathrm{est}_i < \min\{\mathrm{ect}_i : i \in \Theta\}, \mathrm{e}_\Theta + c_i(\min(\mathrm{ect}_i, \mathrm{lct}_\Theta) - \mathrm{est}_\Theta) > \mathcal{C}(\mathrm{lct}_\Theta - \mathrm{est}_\Theta) \Rightarrow \neg(i \prec \Theta) \tag{3.22}$$

$$\max\{\mathrm{lst}_i : i \in \Theta\} < \mathrm{lct}_i, \mathrm{e}_\Theta + c_i(\mathrm{lct}_\Theta - \max(\mathrm{lst}_i, \mathrm{est}_\Theta)) > \mathcal{C}(\mathrm{lct}_\Theta - \mathrm{est}_\Theta) \Rightarrow \neg(\Theta \prec i) \tag{3.23}$$

The reasoning behind (3.22) is that if $S_i = \mathrm{est}_i$, then one can not schedule a task from Θ which completes before i, as $\mathrm{est}_i < \mathrm{ect}_\Theta$. The total energy consumed by all the tasks of $\Theta \cup \{i\}$ in the entire window where Θ can execute is $\mathrm{e}_\Theta + c_i(\min(\mathrm{ect}_i, \mathrm{lct}_\Theta) - \mathrm{est}_\Theta)$, which according to (3.22) causes

the Overload Checking to fail. Therefore, $S_i > \text{est}_i \geq \text{est}_\Theta$ and $\neg(i \prec \Theta)$. A similar reasoning for (3.23) holds.

If (3.22) is detected, the earliest starting time of i is simply adjusted to ect_Θ and if (3.23) is detected, the latest completion time of i is simply adjusted to lst_Θ.

Andreas Schutt, et.al. [72] showed that the Not-First/Not-Last detection algorithm presented by Nuijten [55] which runs in $O(n^3 k)$ is incorrect and incomplete. They presented a new correct and complete detection algorithm running in $O(n^3 \log(n))$. Further, Kameugne et al. [40] present a sound algorithm which utilizes the Θ_L-tree and runs in $O(n^2 \log(n))$.

Not-First/Not-Last in disjunctive scheduling

It turns out that Not-First/Not-Last can be implemented more efficiently for the DISJUNCTIVE constraint. Baptiste and Le Pape [9] present an $O(n^2)$ algorithm to perform all the time bound adjustments and update the starting and ending times of all the tasks. Similar to the Edge-Finding algorithm, Vilím [87] uses the Θ_L-tree to introduce an algorithm in $O(n \log(n))$.

3.6 Energetic Reasoning

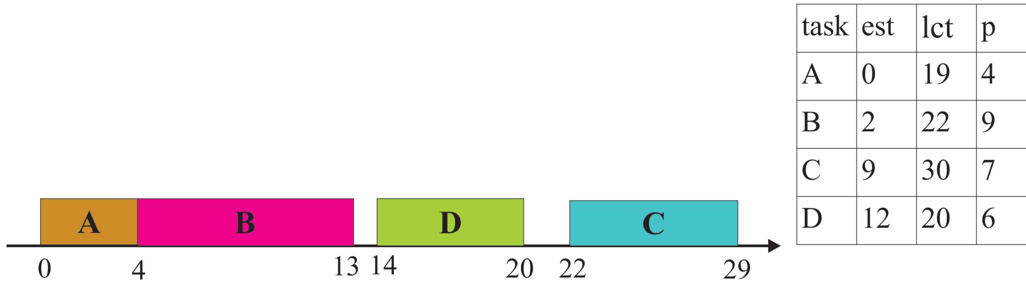
While the filtering techniques presented so far reason about the order of execution of tasks with respect to each other, Energetic Reasoning takes into account the amount of resource energy which is required over an interval $[t_1, t_2)$ with respect to the total amount of energy which is available through the interval. Energetic Reasoning provides the most powerful propagation technique compared to the methods introduced so far. However, it is less studied in scheduling literature, due to its $O(n^3)$ complexity [20]. Recently, there was improvements to $O(n^2 \log(n))$ for the efficiency of this algorithm [79, 13].

If there is not enough energy in the the entire window where two tasks $\{i, j\}$ can execute and all the tasks that partly execute on this window, the Energetic Reasoning deduces a precedence. A task $i \in \mathcal{I}$ with respect to the time interval $[t_1, t_2)$ can take five positions:
(1) i is not processed within $[t_1, t_2)$;
(2) i partly overlaps $[t_1, t_2)$ when it is started before $[t_1, t_2)$ (i.e. i is left-shifted). For instance, a task A for which $\text{dom}(S_A) \in [3, 7]$ and $p_A = 2$, partly overlaps with the interval $[8, 12)$;
(3) i fully fits within $[t_1, t_2)$;
(4) i partly overlaps $[t_1, t_2)$ when it is finished after $[t_1, t_2)$ (i.e. i is right-shifted);
(5) i fully overlaps when it is started before $[t_1, t_2)$ and finished after $[t_1, t_2)$.

With regard to five possible cases mentioned above, the *required energy consumption* of i over $[t_1, t_2)$ is defined:

$$W_i(t_1, t_2) = c_i \text{max}(0, \min(\text{ect}_i - t_1, t_2 - t_1, t_2 - \text{lst}_i, p_i))$$

The sum of the required energy consumption of all tasks except for i is $W_{\neq i}(t_1, t_2)$. Formally, the En-



task	est	lct	p
A	0	19	4
B	2	22	9
C	9	30	7
D	12	20	6

Figure 3.5 – Detectable precedences detects $A \ll C, B \ll C, A \ll D, B \ll D$ for the tasks of table 3.5.

ergetic Reasoning rule detects a precedence if $W_{\neq i}(t_1, t_2) + c_i \cdot \min(t_2 - t_1, \max(0, \min(p_i, \text{ect}_i - t_1))) > \mathcal{C}(t_2 - t_1)$ and adjust est_i to $\max(\text{est}_i, t_2 - (\mathcal{C}(t_2 - t_1) - W_{\neq i}(t_1, t_2)/c_i)$.

Notice that apart from Not-First/Not-Last, the alternative filtering rules presented so far are subsumed by the filtering achieved by Energetic Reasoning.

3.7 Detectable Precedences

Detectable Precedences is merely dedicated to the DISJUNCTIVE constraint. Basically, it captures some precedences which are not detected neither by Edge-Finding nor by Not-First/Not-Last.

Let $i, j \in \mathcal{I}$. If $\text{ect}_i > \text{lst}_j$, then the precedence $j \ll i$ is called *detectable*. Let $\Theta = \{j \in \mathcal{I} : \text{ect}_j > \text{lst}_j\}$. The adjustment rule for est_i is as follows

$$\text{est}_i \leftarrow \max\{\text{est}_i, \text{Env}_{\Theta \setminus \{i\}} / \mathcal{C}\}$$

Figure 3.5 demonstrates the scheduling of a set of tasks with Detectable Precedences $A \ll C$ and $A \ll D$.

Detectable Precedences was presented by Vilím [86] and he later improved it in [82] to obtain an algorithm with a running time complexity of $O(n \log n)$.

3.8 Precedence Graph

Vilím [83] introduces the Precedence Graph, which is a filtering principle for the DISJUNCTIVE constraint. This constraint takes into account all types of possible precedences that can exist among the tasks, including the precedences detected by Edge-Finding and Detectable precedences. The additional precedences include those which are defined by the original problem, the precedences that are added throughout the search and the precedences inferred from combination of precedences which already exist in the problem. All the precedences belong to a set defined as

$$\text{Prec}(i) = \{j \in \mathcal{I} : j \ll i\} \tag{3.24}$$

and the filtering rule for the task i is $\mathrm{est}_i \leftarrow \max\{\mathrm{est}_i, \mathrm{Env}_{\mathrm{Prec}(i)\setminus\{i\}} / \mathcal{C}\}$. The time complexity of the algorithm presented for the Precedence Graph is in $O(n^2)$.

3.9 Combination of filtering techniques

Even though Energetic Reasoning is the strongest filtering algorithm among all the techniques presented in this chapter, the alternative filtering algorithms do not necessarily dominate each other. There exist filtering techniques based on the combinations of techniques presented in this chapter.

Vilím [90] implemented a combination of Edge Finding filtering with the Time-Tabling technique in order to build a stronger algorithm. This algorithm runs in $O(n^2)$. More precisely, he modified the propagation rule of Edge Finding in a way that it could use Time-Tabling to perform a better pruning. Compared to the standard and Extended-Edge-Finding algorithms, the new algorithm needs more iterations in order to reach the fixpoint. The contribution is mostly a trade off between the filtering power and the speed. Likewise, Ouellet et al. [58] presented a Time-Table-Extended-Edge-Finding with a complexity of $O(kn\log(n))$.

Schutt et al. [71] developed a Time-Table-Edge-Finding which performs minimal filtering and rather produces no-goods. Their algorithms and solver obtained the best performances so far. They closed 6 open problems in standard benchmarks.

3.10 Filtering techniques in the state of the art schedulers

One might wonder which one of the techniques introduced in this chapter are used in the state of the art schedulers, such as Choco, Gecode, or IBM ILOG CP Optimizer. It turns out that the algorithms presented are not all used simultaneously. Overload Checking is seldom used and it is rather integrated to the Edge-Finding algorithm. In other words, for the same computation time, the Edge-Finding is able to perform an Overload Checking and in addition, it achieves filtering. The Overload Checking is nevertheless important since it is the first step to obtain an Edge-Finding. Time-Tabling is the cornerstone and the most used technique which appears everywhere. This is mainly due to the fact that its filtering algorithm is fast, it achieves a high level of filtering and it is sufficient to filter all non-candidate solutions. Time-Tabling is implemented in Choco. The Edge-Finding is a prominent filtering technique which must be implemented in combinations with Time-Tabling, as posting the Edge-Finding alone prevents the solver from filtering some assignments that do not satisfy the CUMULATIVE constraint. The Edge-Finding is implemented in Choco and Gecode. The Gecode cumulative propagator uses the combination of Edge-Finding, Overload Checking, and Time-Tabling.

The IBM ILOG Scheduler is a library integrated with C++ to model scheduling problems. In this scheduler there is a parameter, called *enforcement level*, whose value determines which types of filtering rules are combined together. The parameter is identified by certain degrees from low to high. It turns out that at all levels from low to high, Time-Tabling constraint is posted. At the medium level

```
repeat
    repeat
        repeat
            repeat
                if not Overload Checking then
                    fail;
                Detectable Precedences;
            until no more propagation;
            Not-First, Not-Last;
        until no more propagation;
        Edge-Finding;
    until no more propagation;
    Precedence Graph;
until no more propagation;
```

Figure 3.6 – The appropriate sequence for the DISJUNCTIVE constraint, presented in Vilím's thesis [88], which achieves the fixpoint

a combination of DISJUNCTIVE and Time-Tabling is posted and at the higher level a combination of DISJUNCTIVE and Time-Tabling and Edge-Finding is posted. For the enforcement levels, there is a trade-off between the amount of filtering and elapsed time in the sense that the higher levels typically cause more propagation of constraints which results in fewer failures. The reduction of the search space does not always compensate for the time required to run the filtering algorithms. This is the reason why the solver allows to adjust the level of filtering.

Due to their complexities, the alternative filtering algorithms that were introduced in this chapter are not available by default on these solvers, even though they were proven to be useful.

An alternative point to consider is that several filtering algorithms can be combined, as each filtering algorithm removes its own types of inconsistencies. Such a situation entails the computation of a fixpoint, which stands for a state that no more filtering occurs. For the DISJUNCTIVE constraint, Vilím [88] presents the figure 3.6 which shows the most appropriate sequence and achieves the fixpoint.

Finally, it must be mentioned that except for the Overload Checking, all the algorithms described in this chapter are not idempotent. Thereby, in order to achieve the maximum pruning, these filtering algorithms are called multiple times until no more pruning occurs.

Chapter 4

Linear time Algorithms for Disjunctive constraint

Constraint programming offers the advantage of defining constraints that can be reused in a variety of problems. For instance, the DISJUNCTIVE constraint can be used in classical scheduling problems or any context in which two tasks cannot overlap at any time. After introducing the framework and filtering algorithms for the DISJUNCTIVE and CUMULATIVE constraints in the preceding chapters, we are ready to embark on a thorough discussion of these algorithms for the DISJUNCTIVE constraint. This chapter is devoted to tackle this constraint in detail by providing more efficient filtering algorithms.

As noted in the preceding chapters, it is NP-Hard to decide whether the DISJUNCTIVE constraint is satisfiable and therefore it is NP-hard to enforce bounds consistency on this constraint. Nonetheless, the pruning rules such as Time-Tabling, Edge-Finding, etc. can be enforced in polynomial time. Furthermore, when $p_i = 1$, the constraint can be recast in terms of the ALL-DIFFERENT constraint. There exists an $O(n \log(n))$ algorithm for bounds consistency of ALL-DIFFERENT constraint [66], that is based on balanced trees, and it was improved to run in linear time [52, 50] by taking advantage of disjoint set data structures. This algorithm conceived ideas for proposing more efficient algorithms in this chapter. In fact, we reproduce the same scenario where $O(n \log(n))$ filtering algorithms exists for the DISJUNCTIVE constraint based on the $\Theta-$tree and we make these algorithms linear by using disjoint set data structures.

This chapter is an extension of [26] in the sense that we also present experimental results that show the importance of using the right sorting algorithms to achieve linear filtering times for Overload Checking, Time-Tabling, and Detectable Precedences. As pointed out in chapter 3, following [26], Schaus et al. [30] present a linear time time-tabling algorithm. Their algorithm works for the CUMULATIVE constraint while we focus on the DISJUNCTIVE constraint. However, as Schaus et al. explain, the algorithm we present in this chapter achieves more pruning in a single iteration.

Section 4.1 of this chapter explains the preliminary which are used throughout the chapter. The rest of

this chapter presents three new filtering algorithms for the DISJUNCTIVE constraint that that all have a linear running time complexity in the number of tasks. The first algorithm presented in section 4.2 filters the tasks according to the rules of the Time-Tabling. Section 4.3 discusses a new data structure that we introduce and that we call the time line. We discuss the algorithms after we investigate the basic properties of the time line. Section 4.4 takes advantage of the time line to present an algorithm for the Overload Checking that could also be used even for the CUMULATIVE constraint. Section 4.5 introduces an algorithm which enforces the rules of Detectable Precedences. Section 4.6 is devoted to the experiments, verifying that the new algorithms are competitive even for a small number of tasks and outperform existing algorithms as the number of tasks increases. Section 4.7 shows how the time line comes in handy to solve two special scheduling problems in a more efficient manner in polynomial time. Section 4.8 briefly summarizes the contribution of this chapter.

4.1 Preliminaries

Let w be the word-size of the processor and all the time points are encoded with w-bit integers. Assuming that $\mathcal{I}_{\text{est}}$, $\mathcal{I}_{\text{lct}}$, $\mathcal{I}_{\text{ect}}$, $\mathcal{I}_{\text{lst}}$ $\mathcal{I}_p$ denote the ordered set of tasks $\mathcal{I}$, respectively sorted by est, lct, ect, lst, and processing times, all these sets can be sorted in linear time $O(n)$. This assumption is supported by the fact that a word of $w = 32$ bits is sufficient to encode all time points, with a precision of a second, within a period longer than a century. This is sufficient for most industrial applications and an algorithm such as radix sort can sort the time points in time $O(wn)$ which is linear when w is constant.

Furthermore, all the rules that we present in this chapter aim at delaying the earliest starting time of the tasks. To advance the latest completion time, one can create the symmetric problem where task i is transformed into a task i' such that $\text{est}_{i'} = -\text{lct}_i$, $\text{lct}_{i'} = -\text{est}_i$, and $p_{i'} = p_i$. Delaying the earliest starting time in the symmetric problem prunes the latest completion time in the original problem.

4.1.1 Union-Find

The new algorithms we present rely on the Union-Find data structure. The function `UnionFind`(n) initializes n disjoint sets $\{0\}, \{1\}, \ldots, \{n-1\}$ in $O(n)$ steps. The function `Union`(a, b) merges the set that contains element a with the set that contains the element b. The functions `FindSmallest`(a) and `FindGreatest`(a) return the smallest and greatest element of the set that contains a. These three functions run in $O(\alpha(n))$ steps, where α is Ackermann's inverse function. Cormen et al. [16] present how to implement this data structure using trees. The smallest and greatest element of each set can be stored in the root of these trees. This implementation is the fastest in practice. However, we use this data structure in a very specific context where the function `Union`(a, b) is called only when `FindGreatest`$(a) + 1 =$ `FindSmallest`(b). Such a restriction allows to use the Union-Find data structure as presented by Gabow and Tarjan [28] that implements the functions `Union`(a, b), `FindSmallest`(a) and `FindGreatest`(a) in constant time. Although this implementation is the

fastest in theory, it is not so in practice due to a large hidden constant.

4.2 Time-Tabling

As mentioned in chapter 3, the Time-Tabling rule exploits the fact that a task i must execute within its compulsory part. In the disjunctive context, if there exists a task i with a compulsory part and there exists a task j that satisfies $\mathrm{ect}_j > \mathrm{lst}_i$, then consequently j must execute after i. That is,

$$\mathrm{lst}_i < \mathrm{ect}_i \wedge \mathrm{lst}_i < \mathrm{ect}_j \Rightarrow \mathrm{est}'_j = \max(\mathrm{est}_j, \mathrm{ect}_i) \tag{4.1}$$

We present a linear time algorithm that enforces the Time-Tabling rule. Analogous to most of Time-Tabling algorithms, this new algorithm is not idempotent. However, it provides some guaranties on the level of filtering it achieves. Consider the set of compulsory parts $\mathcal{F} = \{[\mathrm{lst}_i, \mathrm{ect}_i) \mid i \in \mathcal{I} \wedge \mathrm{lst}_i < \mathrm{ect}_i\}$ and a task $j \in \mathcal{I}$. The algorithm guarantees that after the filtering occurs, the interval $[\mathrm{est}'_j, \mathrm{ect}'_j)$ does not intersect with any intervals in $\mathcal{F}$. However, the pruning of est_j to est'_j might create a new compulsory part $[\mathrm{lst}_j, \mathrm{ect}'_j)$ that could cause some filtering in a further execution of the algorithm.

The Algorithm 2 proceeds in three steps, each of which is associated to a for loop. The first for loop on line 4 creates the vectors l and u that contain the lower bounds and upper bounds of the compulsory parts. The compulsory parts $[l[0], u[0]), [l[1], u[1]), \ldots, [l[m-1], u[m-1])$ form a sequence of sorted and disjoint semi-open intervals such that each of them is associated to a task i that satisfies $\mathrm{lst}_i < \mathrm{ect}_i$. If two compulsory parts overlap, the algorithm, on line 9, returns *Inconsistent*, as two tasks cannot execute simultaneously in the disjunctive context. When processing the task i that has a compulsory part $[l[k], u[k])$, the algorithm makes sure on line 11, that the task i starts no earlier than $u[k-1]$, so that the tasks that have a compulsory part are all filtered.

The second for loop on line 19 creates a vector r that maps a task i to the compulsory part whose upper bound is the smallest one to be greater than est_i. We therefore have the relation $u[r[i]-1] \leq \mathrm{est}_i < u[r[i]]$.

The third for loop on line 25 filters the tasks that do not have a compulsory part. The tasks are processed by non-decreasing order of processing times. Line 30 checks whether $\mathrm{est}'_i + p_i > l[r[i]]$. If so, then the Time-Tabling rule applies and the new value of est'_i is pruned to $u[c]$. The same task is then checked against the next compulsory part $[l[r[i]+1], u[r[i]+1])$ and so forth. Suppose that a task is filtered both by the compulsory part $[l[c], u[c])$ and the compulsory part $[l[c+1], u[c+1])$. Since we process the tasks by non-decreasing order of processing time, any further task that is filtered by the compulsory part $[l[c], u[c])$ will also be filtered by the compulsory part $[l[c+1], u[c+1])$. The algorithm uses a Union-Find data structure to keep track that these two compulsory parts are *glued* together. The next task j that satisfies $\mathrm{est}'_j + p_j > l[c]$ will be filtered to $u[c+1]$ in a single iteration. The Union-Find data structure can union an arbitrary long sequence of compulsory parts.

Theorem 2. Algorithm 2 enforces the Time-Tabling rule in $O(n)$ steps.

Algorithm 2: `TimeTabling`$(\mathcal{I})$

```
1  m ← 0, k ← 0, l ← [], u ← [], r ← [];
2  est'_i ← est_i, ∀ i ∈ I;
4  for i ∈ I_lst do
5      if lst_i < ect_i then                  // If the task i has a compulsory part
6          if m > 0 then
7              if u[m − 1] > lst_i then
9                  return Inconsistent;
11             else est'_i ← max(est'_i, u[m − 1]);
12
13         l.append(lst_i);
14         u.append(est'_i + p_i);
15         m ← m + 1;
16 if m = 0 then          // Without compulsory parts, no filtering is needed
17     return Consistent;
19 for i ∈ I_est do
20     while k < m ∧ est_i ≥ u[k] do
21         k ← k + 1;
22     r[i] ← k;
23 s ← UnionFind(m);
25 for i ∈ I_p do
26     if ect_i ≤ lst_i then
27         c ← r[i];
28         first_update ← True;
30         while c < m ∧ est'_i + p_i > l[c] do
31             c ← s.FindGreatest(c);
32             est'_i ← max(est'_i, u[c]);
33             if est'_i + p_i > lct_i then
34                 return Inconsistent;
35             if ¬first_update then
36                 s.Union(r[i], c);
37             first_update ← False;
38             c ← c + 1;
39 return Consistent;
```

Proof. Each of the two first for loops iterate through the tasks once and execute operations in constant time. Each time the while loop on line 30 executes more than once, the Union-Find data structure merges two compulsory parts. This can occur at most n times. □

Example 4.2.1. Consider three tasks $\mathcal{I} = \{A, B, C\}$ which are described in the table 4.1. A trace of our algorithm is illustrated in figure 4.1. Tasks A and B have compulsory parts, as indicated in phase I. To schedule the task C, it cannot conflict with the compulsory parts of A and B and it does not fit

between these two tasks, either (phase II). Thus, the first time point which is a candidate to schedule C is 15. The algorithm filters the earliest starting time of C to 15 (phase III) and the Union-Find merges the interval just traversed (phase IV).

task	est	lct	p
A	0	6	5
B	6	19	9
C	2	22	6

Table 4.1 – The information of a set of tasks $\mathcal{I} = \{A, B, C\}$.

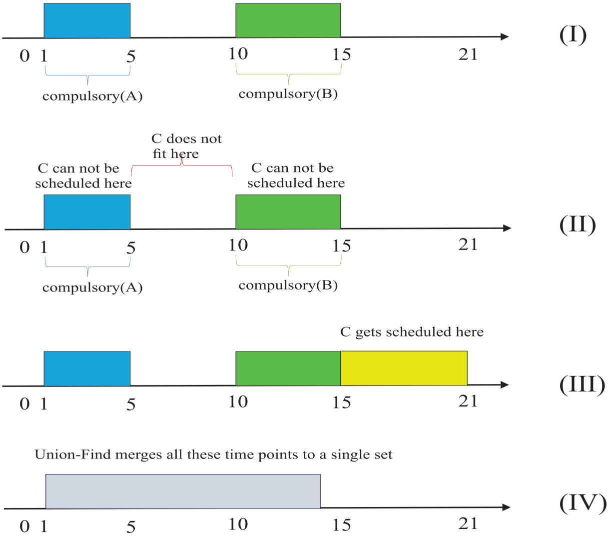


Figure 4.1 – A trace of Time-Tabling algorithm for the tasks of table 4.1.

4.3 The Time Line Data Structure

We introduce a new data structure, which we call it the *time line*. This data structure is initialized with an empty set of tasks $\Theta = \emptyset$. It is possible to add, in constant time, a task to Θ and to compute, in constant time, the earliest completion time ect_Θ. As mentioned in chapter 3, the Θ-tree data structure supports the same operations. However, it differs in two points from the time line in terms of complexity. Inserting a task in Θ-tree requires $O(\log n)$ steps while this operation runs in constant time with the time line. This is the most important novel feature of the time line. Removing a task from a Θ-tree is done in $O(\log n)$ steps and this operation is not efficiently supported by the time line. The time line is therefore faster than Θ-tree, however it can only be used for algorithms where the removal of a task is not required. Table 4.2 shows the advantage of our approach, as well as its limitation compared with the Θ-tree .

Operation	**Θ−tree**	**Time line**
Adding a task to the schedule	$O(\log(n))$	O(1)
Computing the earliest completion time	$O(1)$	$O(1)$
Removing a task from the schedule	$O(\log(n))$	Not supported

Table 4.2 – Comparison of Θ-tree and time line.

The time line data structure is inspired from the one used in the filtering algorithm for the ALL-DIFFERENT constraint designed by López-Ortiz et al. [49]. We consider a sequence $t[0..|t|-1]$ of unique time points sorted in chronological order formed by the earliest starting times of the tasks and a sufficiently large time point, which is $\text{lct}_{\mathcal{I}} + \sum_{i=1}^{n} p_i$. The vector $m[0..n-1]$ maps a task i to the time point index such that $t[m[i]] = \text{est}_i$. The time points, except for the last one, have a capacity stored in the vector $h[0..|t|-2]$. The capacity $h[a]$ denotes the amount of time the resource is available within the semi-open time interval $[t[a], t[a+1])$, should the tasks in Θ be scheduled at their earliest starting time with preemption. Initially, since $\Theta = \emptyset$, the resource is fully available and $h[a] = t[a+1] - t[a]$. A Union-Find data structure s is initialized with $|t|$ elements. This data structure maintains the invariant that a and $a+1$ belong to the same set in s if and only if $h[a] = 0$. This property allows to quickly request, by calling $s.\texttt{FindGreatest}(a)$, the earliest time point no earlier than $t[a]$ with a positive capacity. Finally, the data structure has an index e which is the index of the latest time point whose capacity was decremented. The algorithm 3 initializes the components t, m, h, s, and e that define the time line data structure.

Algorithm 3: `InitializeTimeline`($\mathcal{I}$)

```
1  t ← []
2  h ← []
3  for i ∈ I_est do
4  |   if |t| = 0 ∨ t[|t| − 1] ≠ est_i then
5  |   |   t.append(est_i)
6  |   m[i] ← |t| − 1
7  t.append(max_i lct_i + Σ_{i=1}^{n} p_i)
8  for k = 0..|t| − 2 do
9  |   h[k] ← t[k + 1] − t[k]
10 s ← UnionFind(|t|)
11 e ← −1
```

The data structure allows to schedule a task i over the time line at its earliest time and with preemption. The value $m[i]$ maps the task i to the index of the time point associated to the earliest starting time of task i. Algorithm 4 iterates through the time intervals $[t[m[i]], t[m[i]+1]), [t[m[i]+1], t[m[i]+2]), \dots$ and decreases the capacity of each interval down to 0 until a total of p_i units of capacity is decreased. Each time a capacity $h[k]$ reaches zero, the Union-Find merges the index k with $k+1$ which allows, in the future, to skip arbitrarily long sequences of intervals with null capacities in constant time. This

algorithm returns the completion time of the task which was scheduled.

Algorithm 4: `ScheduleTask`(i)

```
1  ρ ← p_i
2  k ← s.FindGreatest(m[i])
3  while ρ > 0 do
4  |   Δ ← min(h[k], ρ)
5  |   ρ ← ρ − Δ
6  |   h[k] ← h[k] − Δ
7  |   if h[k] = 0 then
8  |   |   s.Union(k, k + 1)
9  |   |   k ← s.FindGreatest(k)
10 e ← max(e, k)
11 return t[k + 1] − h[k]
```

Let Θ be the tasks that were scheduled using the algorithm 4. Algorithm 5 computes in constant time the earliest completion time ect_Θ.

Algorithm 5: `EarliestCompletionTime`(e)

```
1 if e ≥ 0 then
2 |   return t[e + 1] - h[e]
3 else
4 |   return −∞
```

Example 4.3.1. Consider three tasks with parameters as in table 4.3.

task	est	lct	p
A	4	15	5
B	1	10	6
C	5	8	2

Table 4.3 – Parameters of the tasks used to illustrate the scheduling process on the time line.

Initializing the time line produces the structure $\{1\} \stackrel{3}{\to} \{4\} \stackrel{1}{\to} \{5\} \stackrel{23}{\to} \{28\}$ where the numbers in the sets are time points and numbers on the arrows are capacities. After scheduling the task A, the capacity between the time points 4 and 5 becomes null and the Union-Find merges both time points into the same set. The structure changes to $\{1\} \stackrel{3}{\to} \{4, 5\} \stackrel{19}{\to} \{28\}$. After scheduling the task B, the time line changes to $\{1, 4, 5\} \stackrel{16}{\to} \{28\}$ and after scheduling the task C, it becomes $\{1, 4, 5\} \stackrel{14}{\to} \{28\}$. The earliest completion time is given by $28 - 14 = 14$.

Theorem 3. Algorithm 3 runs in $O(n)$ amortized time while Algorithm 4 and Algorithm 5 run in constant amortized time.

Proof. Let h_i be the capacity vector after the ith call to an algorithm among algorithms 3, 4, and 5. We define a potential function $\phi(i) = |\{k \in 0..|t| - 2 \mid h_i[k] > 0\}|$ that is equal to the number of positive components in the vector h_i. Prior to the initialization of the time line data structure, we have $\phi(0) = 0$ since the capacity vector is not even initialized and in all time, we have $\phi(i) \geq 0$. After the initialization, we have $\phi(1) = |t| - 1 \leq n$. The two for loops in Algorithm 3 execute $n + |t| - 1 \leq 2n \in O(n)$ times. Therefore, the amortized complexity of the initialization is $O(n) + \phi(1) - \phi(0) = O(n)$.

Suppose the while loop in Algorithm 4 executes a times. There are at least $a - 1$ and at most a components in the capacity vector that are set to zero, hence $a - 1 \leq \phi(i) - \phi(i-1) \leq a$. The amortized complexity of Algorithm 4 is therefore $a + \phi(i) - \phi(i-1) \leq a - (a-1) \in O(1)$.

Algorithm 5 executes in constant time and does not modify the capacity vector h which implies $\phi(i) = \phi(i-1)$. The amortized complexity is therefore $O(1) + \phi(i) - \phi(i-1) = O(1)$. □

4.4 Overload Checking

The Overload Checking algorithm 1, presented in [89], can be directly used with a time line data structure rather than a Θ-tree. As demonstrated by the algorithm 6, one schedules the tasks, in non-decreasing order of latest completion times. If after scheduling a task i, the algorithm returns an earliest completion time greater than lct_i, then the Overload Checking fails. The total running time complexity of this algorithm is $O(n)$. The proof of correctness is similar to Vilím's algorithm.

Algorithm 6: `OverloadCheck`$(\mathcal{I})$

```
1 InitializeTimeline(I)
2 for j ∈ I_lct do
3     ScheduleTask(j)
4     if EarliestCompletionTime() > lct_j then
5         fail
```

The Overload Checking can be adapted to the CUMULATIVE constraint, as well. One can transform the task i of capacity c_i into a task i' with $\mathrm{est}_{i'} = \mathcal{C}\,\mathrm{est}_i$, $\mathrm{lct}_{i'} = \mathcal{C}\,\mathrm{lct}_i$, $p_{i'} = \mathrm{e}_i$ and $c_{i'} = 1$ The Overload Checking fails on the original problem if and on if it fails on the transformed problem. The transformation preserves the running time complexity of $O(n)$.

4.5 Detectable Precedences

The technique of *Detectable Precedences* consists of finding, for a task i, the set of tasks for which there exists a detectable precedence with i, i.e. $\Omega_i = \{j \in \mathcal{I} \setminus \{i\} \mid \mathrm{ect}_i > \mathrm{lst}_j\}$. Once Ω_i is discovered, one can delay the earliest starting time of i up to ect_{Ω_i}.

$$\mathrm{est}'_i = \max(\mathrm{est}_i, \mathrm{Env}_{\Omega_i} / \mathcal{C}) \tag{4.2}$$

One cannot simply adapt the algorithm in [85] for the time line data structure as it requires to temporarily remove a task among the scheduled tasks which is an operation the time line cannot efficiently do. We circumvent this issue by taking advantage of the time line data structure and designing a new algorithm to enforce the rule of detectable precedences in linear time.

Suppose that the problem has no tasks with a compulsory part, i.e. $\text{ect}_i \leq \text{lst}_i$ for all task $i \in \mathcal{I}$. Algorithm 7 simultaneously iterates over all the tasks i in non-decreasing order of earliest completion times and on all the tasks k in non-decreasing order of latest starting times. Each time the algorithm iterates over the next task i, it iterates (line 9) and schedules (line 12) all tasks k whose latest starting time lst_k is smaller than the earliest completion time ect_i. Once the while loop completes, the set of scheduled tasks is $\{k \in \mathcal{I} \setminus \{i\} \mid \text{lst}_k < \text{ect}_i\}$. We apply the detectable precedence rule by pruning the earliest starting time of task i up to the earliest completion time of the time line (line 21).

Suppose that there exists a task k with a compulsory part, i.e. $\text{ect}_k > \text{lst}_k$. This task could be visited in the while loop before being visited in the main for loop. We do not want to schedule the task k before it is filtered. We therefore call the task k the *blocking task*. When a blocking task k is encountered in the while loop, the algorithm waits to encounter the same task in the for loop. During this waiting period, the filtering of all tasks is postponed. A postponed task i necessarily satisfies the conditions $\text{lst}_k < \text{ect}_i \leq \text{ect}_k$ and $\text{ect}_i < \text{lst}_i$ and therefore the precedence $k \ll i$ holds. When the for loop reaches the blocking task k, it filters the blocking task, schedules the blocking task, and filters the postponed tasks. The blocking task and the set of postponed tasks are reset. It is not possible to simultaneously have two blocking tasks since their compulsory parts would overlap, which is inconsistent with the Time-Tabling rule.

Example 2 Figure 4.2 shows a trace of the algorithm. The for loop on line 7 processes the tasks $\mathcal{I}_{\text{ect}} = \{1, 2, 3, 4\}$ in that order. For the two first tasks 1 and 2, nothing happens: the while loop is not executed and no pruning occurs as no tasks are scheduled on the time line. When the for loop processes task 3, the while loop processes three tasks. The while loop processes the task 2 which is scheduled on the time line. When it processes task 4, the while loop detects that task 4 has a compulsory part in $[14, 18)$, making task 4 the blocking task. Finally, the while loop processes task 1 which is scheduled on the time line. Once the while loop completes, the task 3 is not filtered since there exists a blocking task. Its filtering is postponed until the blocking task is processed. Finally, the for loop processes the task 4. In this iteration, the while loop does not execute. Since task 4 is the blocking task, it is first filtered to the earliest completion time computed by the time line data structure ($\text{est}'_4 \leftarrow 13$). Task 4 is then scheduled on the time line. Finally, the postponed task 3 is filtered to the earliest completion time computed by the time line data structure ($\text{est}'_3 \leftarrow 19$).

Theorem 4. The algorithm `DetectablePrecedences` runs in linear time.

Proof. The for loop on line 7 processes each task only once, idem for the while loop. Finally, a task can be postponed only once during the execution of the algorithm and therefore line 28 is ex-

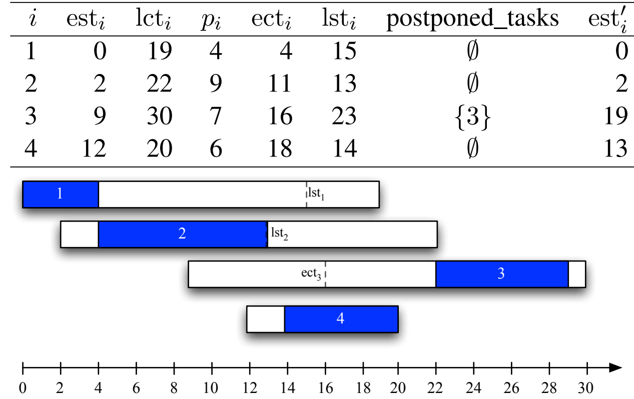
Algorithm 7: `DetectablePrecedences`($\mathcal{I}$)

```
1  InitializeTimeline (𝓘)
2  j ← 0
3  k ← 𝓘_lst[j]
4  postponed_tasks ← ∅
5  blocking_task ← null
7  for i ∈ 𝓘_ect do
9      while j < |𝓘| ∧ lst_k < ect_i do
10         if lst_k ≥ ect_k then
12             ScheduleTask (k)
13         else
14             if blocking_task ≠ null then
15                 return Inconsistent
16             blocking_task ← k
17         j ← j + 1
18         k ← 𝓘_lst[j]
19     if blocking_task = null then
21         est'_i ← max(est_i, EarliestCompletionTime())
22     else
23         if blocking_task = i then
24             est'_i ← max(est_i, EarliestCompletionTime())
25             ScheduleTask (blocking_task)
26             for z ∈ postponed_tasks do
28                 est'_z ← max(est_z, EarliestCompletionTime())
29             blocking_task ← null
30             postponed_tasks ← ∅
31         else
32             postponed_tasks ← postponed_tasks ∪ {i}
33 for i ∈ 𝓘 do
34     est_i ← est'_i
```

ecuted at most n times. Except for `InitializeTimeline` and the sorting of $\mathcal{I}_{\text{ect}}$ and $\mathcal{I}_{\text{lst}}$ that are executed once in $O(n)$ time, all other operations execute in amortized constant time. Therefore, `DetectablePrecedences` runs in linear time. □

4.6 Experimental Results

We experimented our filtering rules with the job-shop and open-shop scheduling problems presented in chapter 2. We model the problems with the starting time variable $S_{i,j}$ for each task j of job i. We post a DISJUNCTIVE constraint over the starting time variables of tasks running on the same machine. For the job-shop scheduling problem, we add the precedence constraints $S_{i,j} + p_{i,j} \leq$



i	est_i	lct_i	p_i	ect_i	lst_i	postponed_tasks	est'_i
1	0	19	4	4	15	$\emptyset$	0
2	2	22	9	11	13	$\emptyset$	2
3	9	30	7	16	23	$\{3\}$	19
4	12	20	6	18	14	$\emptyset$	13

Figure 4.2 – The tasks $\mathcal{I}_{\text{ect}} = \{1, 2, 3, 4\}$ and the visual representation of a solution to the DISJUNCTIVE constraint. The algorithm Detectable Precedences prunes the earliest starting times $\text{est}'_3 = 19$ and $\text{est}'_4 = 13$

$S_{i,j+1}$. For the open-shop scheduling problem, we add a DISJUNCTIVE constraint among all jobs of a task. We use the benchmark provided by [78] that includes 82 and 60 instances of the job-shop and open-shop problems. We implemented our algorithms in Choco 2.1.5 and, as a point of comparison, the Overload Checking and the Detectable Precedences from [82] as well as the Time-Tabling algorithm from [56]. All experiments were carried out on an Intel Xeon X5560 2.667GHz quad-core processor. We used the impact based search heuristic with a timeout of 10 minutes. Each filtering algorithm is individually tested, i.e. we did not combine the filtering algorithms. For the few instances that were solved to optimality within 10 minutes, the two filtering algorithms of the same technique, whether it is Overload Checking, Detectable Precedences, or Time-Tabling, produce the same number of backtracks since they achieve the same filtering. To compare the algorithms, we sum up, for each instance of the same size, the number of backtracks achieved within 10 minutes. In order to statistically check the significancy of our results, we also conducted a two-sample Student's t-Test on all instances at a significance level of 0.05. The null hypothesis of the test is that our results are slower than others. The one-tailed $p-$value of the tests are reported at the last rows of tables 4.4 and 4.5.

Two sorting algorithms

Sorting the tasks with respect to one of their particular parameters is the key in all of the filtering algorithms that we take into account. Normally, one desires to apply an efficient sorting algorithm for this purpose. However, it turns out that in this context there is a much stronger sorting algorithm which can be used. As a matter of fact, since in practice the filtering algorithms are called multiple times during the search process and a constant number of tasks are modified between consecutive calls, one can take advantage of the *insertion sort*, which takes as input the array of almost-sorted tasks as they were sorted on the previous call to the filtering algorithm. When the parameters of only a constant number of tasks are modified, the insertion sort proceeds in linear time [17] and so does the whole

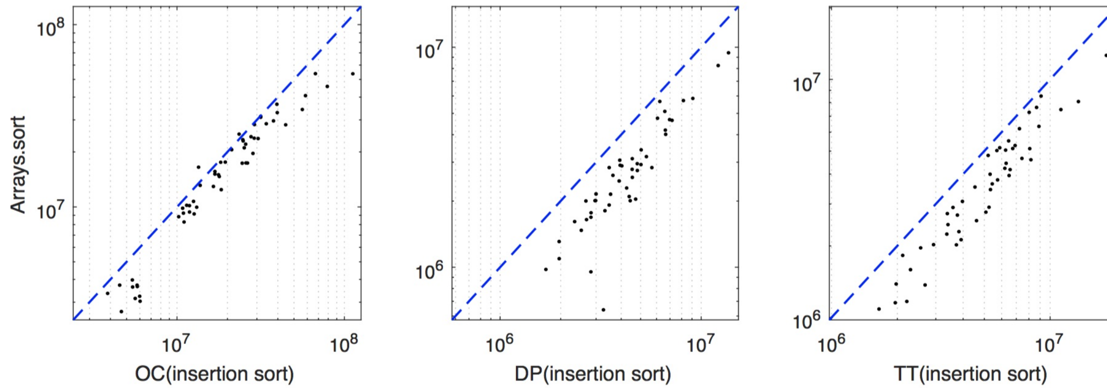


Figure 4.3 – Implementing insertion sort (x axis) and quick sort (y axis) when the state of the art is used for the instances of open shop problem. The figures represent the logarithmic based scale of the number of backtracks occurred within 10 minutes. OC stands for Overload Checking, DP stands for Detectable Precedences and TT stands for Time-Tabling.

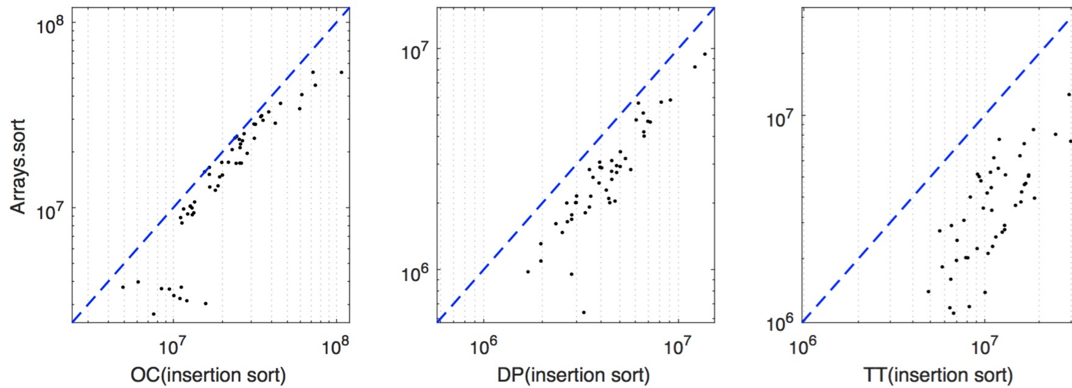


Figure 4.4 – Implementing insertion sort (x axis) and quick sort (y axis) when our algorithms are used for the instances of open shop problem. The figures represent the logarithmic based scale of the number of backtracks occurred within 10 minutes. OC stands for Overload Checking, DP stands for Detectable Precedences and TT stands for Time-Tabling.

filtering algorithm.

This section is devoted to show how insertion sort affects the speed of the algorithms. Therefore, as a point of comparison, we followed two strategies to sort the tasks. We used the function `Arrays.sort` provided by Java 1.7 which is a tuned mergesort implementation and we also implemented the insertion sort as a sorting algorithm which maintains the tasks in the sorted order of the previous call. The figures 4.3 and 4.4 illustrate the performance of insertion sort versus quick sort for the instances of open shop problem, when the data structure used is the state of the art and time line, respectively. As indicated, the insertion sort makes a difference, implying its affect on speeding up the algorithms. Indeed, it speeds up the algorithms, whether we consider state of the art or novel algorithms. The affection is more significant for the state of the art in three different algorithms. Consequently, the fastest sorting algorithm is the insertion sort.

Filtering algorithms

Now that the performance of insertion sort is verified, we compare our filtering algorithms with the state-of-the-art filtering algorithms, both using the insertion sort. We report the ratio of the number of backtracks occured between both algorithms. Note that the two algorithms explore precisely the same search tree in the same order. Thereby, a ratio greater than 1 indicates that our algorithms explore a larger portion of the search tree and therefore they are faster. The tables 4.4 and 4.5 exhibit the results on the open-shop and job-shop problems. Since the DISJUNCTIVE constraint is posted twice on the open shop problem, the rate of the growth in the results of this problem seem to be more uniform, compared to the job shop problem.

The results of the table 4.4 verify that as the size of the instances grows, the new Overload Checking performs much faster.

The p-value obtained for the Overload Checking algorithm in the job shop problem is not conclusive. This might be due to the fact that the initialization of the time line takes longer than the initialization of the $\Theta-$tree.

The new algorithm of Detectable Precedences shows improvements on both problems especially when the number of variables increases. The p-values confirm our hypothesis. One way to explain why the ratios are greater compared with the Overload Checking is that the most costly operations in Vilím's algorithm is the insertion and removal of a task in the Θ-tree which can occur up to 3 times for each task in Detectable Precedences. With the new algorithm, the most costly operation is the scheduling of a task on the time line which occurs only once per task in Overload Checking.

The ratios as well as the student's test also confirm that the new Time-Tabling algorithm is faster. The ratios are higher than with the algorithm by [56], since the latter one was designed for the CUMULATIVE constraint.

Furthermore, we randomly generated large but easy instances with a single DISJUNCTIVE constraint over variables with uniformly generated domains. Unsatisfiable instances and instances solved with zero backtracks were discarded. Table 4.6 shows that the new algorithms are consistently faster.

4.7 Minimizing maximum lateness and total delay

Not only is the time line data structure useful to design filtering algorithms for global scheduling constraints, but also it is strong enough to efficiently solve simple scheduling problems of the form $\alpha|\beta|\gamma$ whose best known algorithm runs in $\Theta(n \log(n))$.

We recall that when minimizing lateness, each task i has a due date $\bar{d}_i$. To our knowledge, the first algorithms for the problems of minimizing maximum lateness and minimizing total delay in the disjunctive and preemptive case were introduced in [39]. These algorithms are roughly similar. To minimize maximum lateness, the algorithm in [39] schedules the tasks, with preemption and at their

$n \times m$	OC	DP	TT
4×4	0.99	1.00	1.00
5×5	1.03	1.00	1.72
7×7	0.99	1.08	1.74
10×10	1.18	1.33	2.31
15×15	1.06	1.22	2.58
20×20	1.82	1.41	2.91
p-value	0.003039719	4.4222E-09	3.61931E-26

Table 4.4 – Open-shop with n jobs and m tasks per job. Ratio of the cumulative number of backtracks between all instances of size $n \times m$ after 10 minutes of computations. OC: our Overload Checking vs. Vilím's. DP: our Detectable Precedences vs Vilím's. TT: Our Time-Tabling vs Ouellet et al. All algorithms use insertion sort.

$n \times m$	OC	DP	TT
10×5	0.94	1.12	1.86
15×5	0.95	1.16	2.07
20×5	0.94	1.37	2.15
10×10	0.95	1.13	2.10
15×10	0.84	1.20	2.06
20×10	0.93	1.34	2.48
30×10	0.95	1.38	2.80
50×10	1.02	1.51	3.29
15×15	0.90	1.14	2.38
20×15	0.89	1.38	2.35
20×20	0.92	1.25	1.70
p-value	0.99	2.07815E-10	1.07719E-23

Table 4.5 – Job-shop with n jobs and m tasks per job. Ratio of the cumulative number of backtracks between all instances of size $n \times m$ after 10 minutes of computations. OC: our Overload Checking vs. Vilím's. DP: our Detectable Precedences vs Vilím's. TT: Our Time-Tabling vs Ouellet et al. All algorithms use insertion sort.

	Overload Checking			Detectable Precedences			Time-Tabling		
n	TT (ms)	TL (ms)	bt	TT (ms)	TL (ms)	bt	Ouellet (ms)	UF (ms)	bt
10	11420	**10716**	142843	7559	**7519**	6803	18652	**15545**	154202
20	7751	**7711**	377305	17311	**14847**	322384	11313	**8902**	140229
30	9606	**9412**	443407	13326	**11109**	136142	11772	**8984**	139346
40	4433	**4112**	5969	19098	**16493**	115986	9551	**7205**	62901
50	5904	**5299**	34454	14895	**12012**	65043	3487	**2871**	3082
60	6150	**5250**	27491	7816	**6952**	3995	6300	**5107**	2612
70	5508	**4737**	17894	5425	**4495**	1514	5505	**3940**	22766
80	28800	**26236**	201453	5915	**4942**	481	2965	**2148**	317
90	31480	**29461**	174305	10016	**7993**	32318	3708	**2939**	509
100	48686	**46104**	262883	9879	**8156**	2360	7393	**5564**	1190

Table 4.6 – Random instances with n tasks. Times are reported in milliseconds. Algorithms implementing the same filtering technique lead to the same number of backtracks (bt). TT: $\Theta-$tree, TL: time line, UF:Union-Find data structure.

earliest time, in non-decreasing order of due dates. To minimize total delay, the algorithm schedule the tasks in non-decreasing order of processing times.

It turns out that the time line provides a more efficient way to master these algorithms.

Algorithm 8: `MinimizingMaximumLateness`($\mathcal{I}$)

1 `InitializeTimeline` ($\mathcal{I}$)
2 **for** $i \in \mathcal{I}_{\bar{d}}$ **do**
3 $\quad E_i \leftarrow$ `ScheduleTask`(i)
4 $\quad L_i \leftarrow E_i - d_i$
5 $L_{\max} \leftarrow \max_{i \in \mathcal{I}}(L_i)$
6 **return** $L_{\max}$

Algorithm 9: `MinimizingTotalDelay`($\mathcal{I}$)

1 `InitializeTimeline` ($\mathcal{I}$)
2 **for** $i \in \mathcal{I}_p$ **do**
3 $\quad E_i \leftarrow$ `ScheduleTask`(i)
4 $\quad D_i \leftarrow E_i - \mathrm{est}_i$
5 $D \leftarrow \sum_{i \in \mathcal{I}} D_i$
6 **return** D

It is proven that the algorithms 8 and 9 minimize the maximum lateness and total delays [39]. Since scheduling the tasks runs in constant time, the overall complexity of these algorithms is linear.

4.8 Conclusion

We presented new filtering algorithms that enforce existing filtering rules. Although these algorithms achieve the same level of filtering achieved as the state-of-the-art algorithms, they have running time complexities that are smaller by a factor $\log(n)$ over the best algorithms enforcing the same filtering techniques. This gain is mostly due to the design of time line data structure that allows to achieve simple computations in constant time rather than in $O(\log(n))$ time. Time line also allows to solve faster two classical problems, namely the minimizing of maximum lateness and total delay.

Chapter 5

Overload Checking and Edge-Finding for Robust Cumulative Scheduling

In the preceding chapter we outlined the filtering techniques in a determinist context. In reality, the scheduling of tasks can be subject to disruptive behaviours that are caused by undesirable factors. In such contexts, the execution of tasks takes longer than expected. Inevitably, replanning is necessary. *Robust project scheduling* is primarily concerned with such unruly environments in the cumulative context.

The material in this chapter follows the framework provided by Derrien et al. [21] for robust cumulative scheduling, where they assume that for a set of n tasks $\mathcal{I}$, at most r of them can be delayed at the same time, without the requirement to reschedule the alternative tasks. The considered stochastic framework can capture practical problems, such as the crane assignment problem [21]. In this problem, the planner needs a schedule which still respects the deadlines in case at most r tasks are delayed. Derrien et.al present the adaption of Time-Tabling algorithm. Even though they define the paradigm for r delayed tasks, they focus on the case $r = 1$. This chapter considers any $r > 0$ and adapts the Overload Checking [89] and Edge-Finding [84] for this framework.

The rest of this chapter is organized as follows. Section 5.1 describes the terms and notations which are most frequently used throughout the chapter. It also presents the FLEXC constraint as a variation of CUMULATIVE constraint in robust contexts. Section 5.2 presents the robust algorithm of Overload Checking. Section 5.3 is devoted to the Edge-Finding algorithm for filtering the domains of the starting time variables. Section 5.4 presents and discuss the experiments. In section 5.5 we conclude.

5.1 Preliminaries and the general framework

In this section, we describe the terminology specialized for the framework considered in this chapter.

Due to some considerable uncertainties, one can not absolutely trust the task durations, as the process-

ing of tasks can take longer than expected. Such a situation entails the assignment of an attribute d_i known as the *delay* duration to each task i. From that, a task i has two extra parameters, the delayed processing time $p_i^d = p_i + d_i$ and the delayed latest completion time $\mathrm{lct}_i^d = \mathrm{lct}_i + d_i$. Furthermore,

$$\mathrm{lct}_\Theta^d = \max\{\mathrm{lct}_i^d \mid i \in \Theta\} \tag{5.1}$$

We refer to the tasks when they are delayed or not delayed with particular symbols. A task i that is not delayed is called *regular* and it is denoted i^0 and a task i that is delayed is denoted i^1. Throughout this chapter, we associate a regular task with 0 and a delayed task with 1. For instance, the set $\{A^0, B^1, C^0, D^1\}$ is a set of tasks where A and C are not delayed and B and D are delayed. $\mathcal{I}^0 = \{i^0 \mid i \in \mathcal{I}\}$ refers to the regular tasks from $\mathcal{I}$ and $\mathcal{I}^1 = \{i^1 \mid i \in \mathcal{I}\}$ refers to the delayed tasks from $\mathcal{I}$. Moreover, for $i^1 \in \mathcal{I}^1$, the processing time p_i^d is considered and for $i^0 \in \mathcal{I}^0$, the processing time p_i is considered. For $i \in \mathcal{I}$ and $b \in \{0, 1\}$, we define $\mathrm{e}_{i^b} = p_i^b c_i$. For a subset $\Theta \subseteq \mathcal{I}^0 \cup \mathcal{I}^1$, we define

$$\mathrm{e}_\Theta = \sum_{i^b \in \Theta} \mathrm{e}_{i^b} \tag{5.2}$$

For a subset $\Theta \subseteq \mathcal{I} \cup \mathcal{I}^0 \cup \mathcal{I}^1$,

$$\mathcal{I}(\Theta) = \{i \in \mathcal{I} \mid i^0 \in \Theta \vee i^1 \in \Theta \vee i \in \Theta\} \subseteq \mathcal{I} \tag{5.3}$$

Note that $\mathcal{I}(\Theta)$ is the set of tasks from $\mathcal{I}$, whatsoever. That is, the delay status of the tasks is not specified in $\mathcal{I}(\Theta)$.

5.1.1 Robust cumulative constraint

Derrien et.al [21] introduce the Robust Cumulative Problem of order r (RCuSP^r) by integrating the notion of robustness to CuSP. According to this framework, a set $\Theta^1 \subseteq \mathcal{I}^1$ of at most $r \geq 1$ tasks can be delayed up to their associated delay attribute without shifting the position of other tasks. A solution to RCuSP^r satisfies

$$\forall t, \forall \Theta^1 \subseteq \mathcal{I}^1, |\Theta^1| \leq r : \sum_{i \in \mathcal{I} : S_i \leq t < S_i + p_i} c_i + \sum_{j^1 \in \Theta^1 : S_j + p_j \leq t < S_j + p_j^d} c_j \leq \mathcal{C} \tag{5.4}$$

The first summation in constraint (5.4) adds the capacities of all tasks executing at time t and the second summation adds the capacities of at most r tasks whose delayed part intersects with time t. We refer to the constraint (5.4) by
FLEXC$([S_1, \ldots, S_n], [p_1, \ldots, p_n], [d_1, \ldots, d_n], [c_1, \ldots, c_n], \mathcal{C}, r)$.
Let

$$1(x) = \begin{cases} 1 & \text{if } x \text{ is true} \\ 0 & \text{if } x \text{ is false} \end{cases} \tag{5.5}$$

The following relation indicates that this problem considers $\binom{n}{r}$ scenarios where r tasks among n are delayed and the CUMULATIVE constraint holds no matter which of r tasks are delayed.

$$\textsc{FlexC}([S_1,\ldots,S_n],[p_1,\ldots,p_n],[d_1,\ldots,d_n],[c_1,\ldots,c_n],\mathcal{C},r) \iff$$

$$\bigwedge_{\substack{\Theta^1 \subseteq \mathcal{I}^1 \\ |\Theta^1|=r}} \textsc{Cumulative}([S_1,...,S_n],[p_1^{1(1\in\Theta^1)},...,p_n^{1(n\in\Theta^1)}],[c_1,...,c_n],\mathcal{C}) \tag{5.6}$$

The algorithms that we adapt to cover delayed tasks efficiently emulates the state of the art algorithms on the conjunction of CUMULATIVE constraints (5.6).

5.2 Robust Overload Checking

The objective of this section is to adapt the Overload Checking algorithm 1 for the $\textsc{FlexC}([S_1,\ldots,S_n],[p_1,\ldots,p_n],[d_1,\ldots,d_n],[c_1,\ldots,c_n],\mathcal{C},r)$ constraint. In section 5.2.1 we establish the generic form of the robust Overload Checking rule and illustrate it with an example. Section 5.2.2 introduces the notion of earliest energy envelope of a set of tasks in the robust context. Section 5.2.3 recasts the robust Overload Checking rule in terms of the robust energy envelope. Afterwards, in section 5.2.4 we propose an extended data structure which enables us to handle the tasks when a certain number of them are delayed. Finally, in section 5.2.5 we take advantage of the introduced data structure to present the robust Overload Checking algorithm. Moreover, we trace few steps of the algorithms for the example provided. This section finishes with a discussion on the time complexity of the algorithm.

5.2.1 The general form of robust Overload Checking rule

Let $\Theta^0 \subseteq \mathcal{I}^0, \Theta^1 \subseteq \mathcal{I}^1$, such that $|\Theta^1| \le r$ and $\mathcal{I}(\Theta^0) \cap \mathcal{I}(\Theta^1) = \emptyset$. The Overload Checking triggers a failure if

$$\mathcal{C}(\max(\text{lct}_{\mathcal{I}(\Theta^0)}, \text{lct}^d_{\mathcal{I}(\Theta^1)}) - \text{est}_{\mathcal{I}(\Theta^0\cup\Theta^1)}) < \text{e}_{\Theta^0} + \text{e}_{\Theta^1} \tag{5.7}$$

Example 5.2.1. Consider the table 5.1 which corresponds to a set of tasks $\mathcal{I} = \{A,B,C,D,E\}$ that must execute on a resource of capacity $\mathcal{C} = 4$ in the context where at most $r = 2$ tasks are allowed to delay. Let $\Theta = \{A,B,C,D\} \subseteq \mathcal{I}$. If $\Theta^0 = \{A^0, C^0\}$ and $\Theta^1 = \{B^1, D^1\}$, the Overload Checking

task	est	lct	p	d	c
A	1	20	5	1	2
B	4	23	7	2	4
C	0	14	4	0	3
D	0	21	8	3	4
E	9	26	2	1	1

Table 5.1 – A set of tasks $\mathcal{I} = \{A,B,C,D,E\}$ to execute on a resource of capacity $\mathcal{C} = 4$. The Overload Checking fails according to (5.7) for $\Theta = \{A,B,C,D\} \subseteq \mathcal{I}, \Theta^0 = \{A^0,C^0\}$ and $\Theta^1 = \{B^1,D^1\}$.

returns a failure, for

$$100 = 4(25-0) < (10+12) + (36+44) = 102$$

$\square$

5.2.2 Robust earliest energy envelope

One can generalize the notion of the earliest energy envelope when tasks can be delayed. Rather than computing the earliest energy envelope of a set Θ as in (3.2), we compute the earliest energy envelope of two sets $\Theta^0 \subseteq \mathcal{I}^0$ and $\Theta^1 \subseteq \mathcal{I}^1$. The tasks in Θ^0 can be regular while the tasks in Θ^1 can be delayed. The tasks that belong to both sets can either be regular or delayed and not both. The earliest energy envelope for the sets Θ^0 and Θ^1 in such a case is defined

$$\mathrm{Env}^r(\Theta^0, \Theta^1) = \max_{\substack{\Omega^0 \subseteq \Theta^0 \\ \Omega^1 \subseteq \Theta^1 \\ |\Omega^1| \leq r \\ \mathcal{I}(\Omega^0) \cap \mathcal{I}(\Omega^1) = \emptyset}} (\mathcal{C}\,\mathrm{est}_{\mathcal{I}(\Omega^0 \cup \Omega^1)} + \mathrm{e}_{\Omega^0} + \mathrm{e}_{\Omega^1}) \tag{5.8}$$

Once (5.8) is computed, a lower bound for the earliest completion time of $\Theta^0 \cup \Theta^1$ is obtained by $\lceil \mathrm{Env}^r(\Theta^0, \Theta^1)/\mathcal{C} \rceil$.

5.2.3 Robust Overload Checking rule in terms of robust energy envelope

We establish an equivalent criterion for the Overload Checking rule (5.7) in terms of $\mathrm{Env}^r(\Theta^0, \Theta^1)$. Let $T = \{\mathrm{lct}_i : i \in \mathcal{I}\} \cup \{\mathrm{lct}_i^d : i \in \mathcal{I}\}$ be the set of all latest completion time and delayed latest completion time points. For $t \in T$, let $\mathrm{Lcut}^0(t) = \{i^0 \in \mathcal{I}^0 : \mathrm{lct}_i \leq t\}$ be the *regular left cut* of t and $\mathrm{Lcut}^1(t) = \{i^1 \in \mathcal{I}^1 : \mathrm{lct}_i^d \leq t\}$ be the *delayed left cut* of t. The former signifies the set of regular tasks which terminate no later than t and the latter refers to the set of delayed tasks which terminate no later than t. Similar to lemma 1 in chapter 3, the following lemma provides a criterion in robust context for checking e-feasibility according to the envelope of the tasks scheduled.

Lemma 2. The Overload Checking fails according to the rule (5.7) if and only if for some $t \in T$

$$\mathrm{Env}^r(\mathrm{Lcut}^0(t), \mathrm{Lcut}^1(t)) > \mathcal{C} \cdot t$$

Proof. Consider $\Theta^0 \subseteq \mathcal{I}^0$ and $\Theta^1 \subseteq \mathcal{I}^1$ with $|\Theta^1| \leq r$ and $\mathcal{I}(\Theta^0) \cap \mathcal{I}(\Theta^1) = \emptyset$ as the subsets for which (5.7) implies that the Overload Checking fails and let $\max(\mathrm{lct}_{\mathcal{I}(\Theta^0)}, \mathrm{lct}^d_{\mathcal{I}(\Theta^1)}) = \mathrm{lct}_{\mathcal{I}(\Theta^0)}$ in (5.7). If j corresponds to a task $j \in \mathcal{I}(\Theta^0)$ for which $\mathrm{lct}_j = \mathrm{lct}_{\mathcal{I}(\Theta^0)}$ and by setting $t = \mathrm{lct}_j$, the assumption for the selected task j implies $\Theta^0 \subseteq \mathrm{Lcut}^0(t)$ and $\Theta^1 \subseteq \mathrm{Lcut}^1(t)$. From (5.7) it follows that

$$\mathcal{C} \cdot t = \mathcal{C} \cdot \mathrm{lct}_{\mathcal{I}(\Theta^0)} < \mathcal{C} \cdot \mathrm{est}_{\mathcal{I}(\Theta^0 \cup \Theta^1)} + \mathrm{e}_{\Theta^0} + \mathrm{e}_{\Theta^1} \leq \mathrm{Env}^r(\mathrm{Lcut}^0(t), \mathrm{Lcut}^1(t))$$

A similar reasoning holds if $\max(\mathrm{lct}_{\mathcal{I}(\Theta^0)}, \mathrm{lct}^d_{\mathcal{I}(\Theta^1)}) = \mathrm{lct}^d_{\mathcal{I}(\Theta^1)}$.

Now, assume that for some $t \in T$, $\mathrm{Env}^r(\mathrm{Lcut}^0(t), \mathrm{Lcut}^1(t)) > \mathcal{C} \cdot t$. Let $\Theta^0 \subseteq \mathrm{Lcut}^0(t)$, $\Theta^1 \subseteq \mathrm{Lcut}^1(t)$ be the subsets for which

$$\mathrm{Env}^r(\mathrm{Lcut}^0(t), \mathrm{Lcut}^1(t)) = \mathcal{C}\,\mathrm{est}_{\mathcal{I}(\Theta^0 \cup \Theta^1)} + \mathrm{e}_{\Theta^0} + \mathrm{e}_{\Theta^1}$$

where $\left|\Theta^1\right| \leq r$ and $\mathcal{I}(\Theta^0) \cap \mathcal{I}(\Theta^1) = \emptyset$. Therefore,

$$\mathcal{C} \cdot \max(\text{lct}_{\mathcal{I}(\Theta^0)}, \text{lct}^d_{\mathcal{I}(\Theta^1)}) \leq \mathcal{C} \cdot t < \text{Env}^r(\text{Lcut}^0(t), \text{Lcut}^1(t)) = \mathcal{C}\,\text{est}_{\mathcal{I}(\Theta^0 \cup \Theta^1)} + \text{e}_{\Theta^0} + \text{e}_{\Theta^1}$$

which implies (5.7). □

In the following, we present a data structure from which $\text{Env}^r(\Theta^0, \Theta^1)$ can be retrieved efficiently.

5.2.4 Θ^r_L-tree

While maintaining the same structure for the $\Theta-$tree, introduced in chapter 3, we extend it to a tree, called Θ^r_L-tree, by adding new parameters to the nodes to handle the case where at most r tasks could be delayed. In the Θ^r_L-tree, two sets of task $\Theta^0 \subseteq \mathcal{I}^0$ and $\Theta^1 \subseteq \mathcal{I}^1$ affect the values of the parameters in the leaves and therefore in the entire tree. In what follows, the symbols $\text{e}^0_v, \text{Env}^0_v$ and $\text{e}^1_v, \text{Env}^1_v$ stand for the energy and the earliest energy envelope of a task v whether this task is regular or delayed. They are computed as

$$\text{e}^k_v = \begin{cases} c_v p_v & \text{if } (v^0 \in \Theta^0) \wedge (k = 0 \vee v^1 \notin \Theta^1) \\ c_v p^d_v & \text{if } (v^1 \in \Theta^1) \wedge (k > 0) \\ 0 & \text{otherwise} \end{cases} \tag{5.9}$$

$$\text{Env}^k_v = \begin{cases} \mathcal{C}\,\text{est}_v + \text{e}^k_v & \text{if } (v^0 \in \Theta^0) \vee (v^1 \in \Theta^1) \\ -\infty & \text{otherwise} \end{cases} \tag{5.10}$$

The superscript k stands for an upper bound on the number of delayed tasks in the leaf v. Since the task corresponding to v is either regular or delayed, only two cases for $k(k \in \{0, 1\})$ make sense. However, in order to make the computations of the energy and envelope symmetrical over all nodes of the tree, we suppose that $0 \leq k \leq r$ and since e^k_v and Env^k_v are the energy and earliest energy envelope when at most k tasks are delayed, for $k \geq 2$ in the leaves of the Θ^r_L-tree we necessarily have $\text{e}^k_v = \text{e}^1_v$ and $\text{Env}^k_v = \text{Env}^1_v$.

Let w be an internal node of the tree. The symbols e^r_w and Env^r_w stand for the energy and envelope of all tasks associated to the leaves that are descendants of w. Moreover, among these tasks, a most r tasks in Θ^1 can be delayed. If the left and right children of w are respectively denoted left(w) and right(w), according to (3.6) and (3.7)

$$\text{e}^0_w = \text{e}^0_{\text{left}(w)} + \text{e}^0_{\text{right}(w)} \tag{5.11}$$

$$\text{Env}^0_w = \max(\text{Env}^0_{\text{left}(w)} + \text{e}^0_{\text{right}(w)}, \text{Env}^0_{\text{right}(w)}) \tag{5.12}$$

Scheduling a regular or delayed task is equivalent to adding the task i^0 to Θ^0 or the task i^1 to Θ^1 by updating the node corresponding to the task according to (5.9) and (5.10). Afterwards, the values of

the energy and envelope for the set of all tasks which are inserted in the tree so far can be recursively computed for the internal nodes of the tree. For at most k delayed tasks the contribution of at most j delayed tasks in the left and at most $k-j$ tasks in the right subtree emanating from w, $0 \leq j \leq k$, must be added up. Hence the energies of all tasks where at most k tasks are delayed, denoted e_w^k, and the sum of the earliest energy envelope of all tasks where at most k tasks are delayed, denoted Env_w^k, $0 \leq k \leq r$, are recursively computed as

$$\mathrm{e}_w^k = \max_{0 \leq j \leq k}\{\mathrm{e}_{\text{left}(w)}^j + \mathrm{e}_{\text{right}(w)}^{k-j}\} \tag{5.13}$$

$$\mathrm{Env}_w^k = \max_{0 \leq j \leq k}\{\mathrm{Env}_{\text{left}(w)}^j + \mathrm{e}_{\text{right}(w)}^{k-j}\} \cup \{\mathrm{Env}_{\text{right}(w)}^k, \mathrm{Env}_w^{k-1}\} \tag{5.14}$$

In the case that the number of delayed task is greater than the number of available nodes in the subtree of the right side, it is sufficient to retrieve Env_w^{k-1}, i.e. the energy envelope for when at most $k-1$ tasks are delayed. At the root of the Θ_L^r-tree, we obtain $\mathrm{Env}_{\text{root}}^r = \mathrm{Env}^r(\Theta^0, \Theta^1)$. This quantity is essential to perform the Overload Checking.

Lemma 3. The update of the Θ_L^r-tree runs in $\Theta(r^2 \log(n))$.

Proof. Upon the update of the values e_v^k and Env_v^k of a leaf as well as all nodes along the path connecting the leaf node to the root of the Θ_L^r-tree, there are r functions e_v^k to compute in constant time. There are also r functions Env_v^k, each one computed in $O(r)$ time, which deduces $\Theta(r^2 \log(n))$ computations. □

5.2.5 Robust Overload Checking algorithm

Let t_1 and t_2 be two arbitrary time points such that $t_1 < t_2$. The Overload Checking test ensures that the total energy of the tasks executing within the interval $[t_1, t_2]$ does not exceed the total energy available inside it. For the rule (3.1) it turns out that it suffices to check it for $t_1 = \mathrm{est}_i, t_2 = \mathrm{lct}_j$ for $i, j \in \mathcal{I}$. In the algorithm that we propose, t_2 could also be $t_2 = \mathrm{lct}_j^d$ for $j^1 \in \mathcal{I}^1$.

In order to develop the Overload Checking rule in the robust context, let $T = \{\mathrm{lct}_i : i \in \mathcal{I}\} \cup \{\mathrm{lct}_i^d : i \in \mathcal{I}\}$ be the set of all latest completion time and delayed latest completion time points. The algorithm starts with an empty Θ_L^r-tree, i.e. $\Theta^0 = \Theta^1 = \emptyset$. That is, no tasks is initially scheduled. The idea is to process the time points $t \in T$ in non-decreasing order and schedule the regular tasks whose latest completion time is equal to t by adding them to Θ^0 or schedule the delayed tasks whose delayed latest completion time is equal to t by adding them to Θ^1. Such an addition changes the energy and envelopes for the leaves corresponding to the tasks in the tree as well as for all nodes on the branch connecting the leaves to the root. At each iteration, if a task i satisfies $\mathrm{lct}_i = t$, then it is scheduled on the Θ_L^r-tree and i^0 is added to Θ^0 and if $\mathrm{lct}_i^d = t$, then i is updated in the Θ_L^r-tree and i^1 is added to Θ^1. Once scheduling the tasks corresponding to t is over, the Overload Checking rule (5.7) must be assessed. Thanks to lemma 2, in order to assess (5.7) in the algorithm, it suffices to check

Algorithm 10: Overload Checking($\mathcal{I}$, $\mathcal{C}$,r)

1 $\Theta^0 \leftarrow \emptyset$
2 $\Theta^1 \leftarrow \emptyset$
3 $T \leftarrow \{\text{lct}_i : i \in \mathcal{I}\} \cup \{\text{lct}_i^d : i \in \mathcal{I}\}$
4 **for** $t \in T$ *sorted in non-decreasing order* **do**
5 $\quad \Theta^0 \leftarrow \Theta^0 \cup \{i \in \mathcal{I} : \text{lct}_i = t\}$
6 $\quad \Theta^1 \leftarrow \Theta^1 \cup \{i \in \mathcal{I} : \text{lct}_i^d = t\}$
7 $\quad$ **if** $\text{Env}^r(\Theta^0, \Theta^1) > \mathcal{C} \cdot t$ **then**
8 $\quad\quad$ **fail**

$\text{Env}^r(\text{Lcut}^0(t), \text{Lcut}^1(t)) > \mathcal{C} \cdot t$ for $t \in T$ which is being processed and $\text{Env}^r(\text{Lcut}^0(t), \text{Lcut}^1(t))$ can be retrieved from the root of the Θ_L^r-tree that is developed so far. The algorithm 10 implements the Overload Checking algorithm in the robust context.

In order to elucidate the mechanism of the algorithm 10, we present few steps of implementing the algorithm for the example 5.2.1 and illustrate them in figure 5.1. Note that the nodes of the Θ_L^r-tree that are affected during the updates that occur at each step are discriminated in blue colours. For this example we have $\mathcal{I} = \{A, B, C, D, E\}$ and $T = \{14, 20, 21, 23, 24, 25, 26, 27\}$. After initializing an empty Θ_L^r-tree, in the first iteration the regular tasks i^0 for which $\text{lct}_i = 14$ and the delayed tasks i^1 for which $\text{lct}_i^d = 14$ are scheduled. C is the only tasks which qualifies in both conditions. Therefore, C^0 is added to Θ^0 and C^1 is added to Θ^1. The figure 5.1a depicts the status of the Θ_L^r-tree after this iteration. In the next iteration, A is the only eligible task to be scheduled, but it is not allowed to be delayed, as $\text{lct}_A = 20$ and $\text{lct}_A^d = 21 > t = 20$. Therefore A^0 is added to Θ^0 but A^1 is not yet added to Θ^1. The figure 5.1b illustrates the status of the Θ_L^r-tree after this iteration. In the next iteration, corresponding to the figure 5.1c, A^1 and D^0 are scheduled, as $\text{lct}_A^d = \text{lct}_D = 21$. The figure 5.1d demonstrates the status of the Θ_L^r-tree at iteration $t = 25$. Before $t = 25$ is processed at this step $\Theta^0 = \{C^0, A^0, D^0, B^0\}$ and $\Theta^1 = \{C^1, A^1, D^1\}$. Since $\text{lct}_B^d = 25$, B^1 must be added to Θ^1. The insertion of B^1 into the Θ_L^r-tree causes the Overload Checking to fail, as $\text{Env}^2(\Theta^0, \Theta^1) = \text{Env}^2_{\text{root}} = 102 > 4 \cdot 25 = 100$.

Lemma 4. The algorithm 10 runs in $\Theta(r^2 n \log(n))$.

Proof. According to lemma 3, the lines 5 and 6 of the algorithm 10 run in $\Theta(r^2 \log(n))$. Since every task is inserted exactly once in Θ^0 and once in Θ^1, overall the computational effort is $\Theta(r^2 n \log(n))$. □

5.3 Robust Edge-Finding

The objective of this section is to adapt the Edge-Finding algorithm, introduced in section 3.3.1, for the FLEXC$([S_1, \ldots, S_n], [p_1, \ldots, p_n], [d_1, \ldots, d_n], [c_1, \ldots, c_n], \mathcal{C}, r)$ constraint. We present a gener-

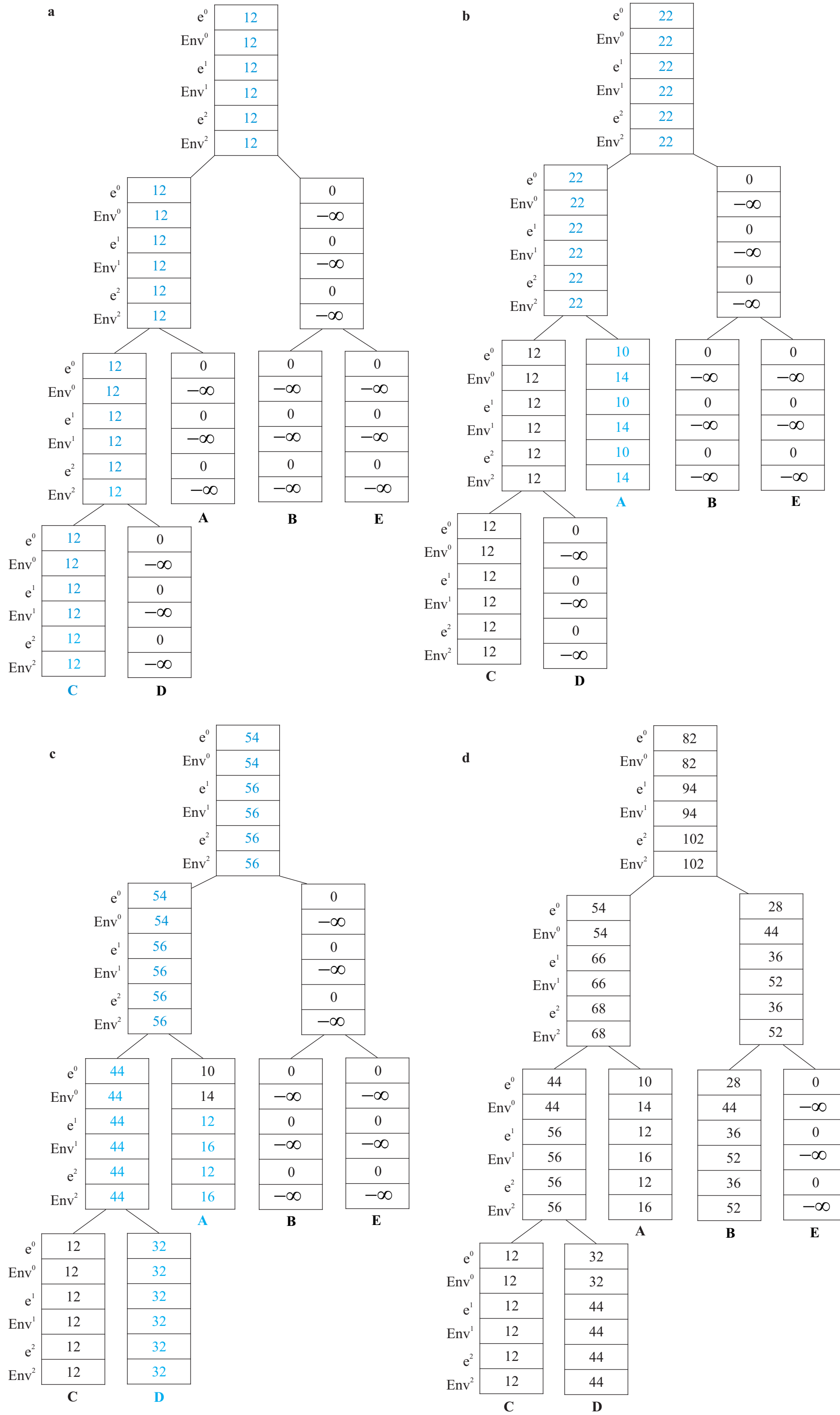


Figure 5.1 – In the picture (a), C^0 and C^1 are scheduled in the Θ_L^r−tree. The coloured nodes in blue represent the affected nodes during the update of the tree. In this case, $\Theta^0 = \{C^0\}$ and $\Theta^1 = \{C^1\}$. In the picture (b), A^0 is scheduled in the Θ_L^r−tree, but not A^1. In this case, $\Theta^0 = \{C^0, A^0\}$ and $\Theta^1 = \{C^1\}$. In the picture (c), A^1 and D are scheduled. In this case, $\Theta^0 = \{C^0, A^0, D^0\}$ and $\Theta^1 = \{C^1, A^1\}$. Picture (d) represents the status of the Θ_L^r−tree when processing $t = 25$. Since $\text{Env}^2(\Theta^0, \Theta^1) = \text{Env}^2_{\text{root}} = 102 > 4 \cdot 25 = 100$ the Overload Checking triggers a failure.

alization of the Edge-Finding detection rules (3.10) and (3.11) for the
$\textsc{FlexC}([S_1,\ldots,S_n],[p_1,\ldots,p_n],[d_1,\ldots,d_n],[c_1,\ldots,c_n],\mathcal{C},r)$ constraint. Prior to expound the mechanism of our algorithms, we should be mindful that in contrast to the regular scheduling problems, where the tasks are not assumed to delay, the robust framework is not symmetrical [21]. Indeed, a task can be delayed but it cannot be brought forward. Therefore, one can not simply apply the rule of filtering earliest starting times to the symmetrically negated problem, as we also did in chapter 4, in order to filter latest completion times. Therefore, we treat the filtering of lower bounds and upper bounds independently.

The structure of this section is as follows. Section 5.3.1 is concerned with filtering the earliest starting times. First we establish the generic form of the robust Edge-Finding rule and accompany that with an example. Then, a new variant of robust earliest energy envelope for the conjunction of two disjoint subsets of tasks is defined. Afterwards, we propose an extended data structure which enables us to handle the tasks when a certain number of them are delayed. Finally, we take advantage of the introduced data structure to present the robust Edge-Finding algorithm. Moreover, we trace few steps of the algorithms for the example provided. We finish this part with a discussion on the time complexity of the algorithm. Section 5.3.2 studies the filtering of latest completion times and follows the same structure as Section 5.3.1.

5.3.1 Filtering the earliest starting times

This section discusses the Edge-Finding rule, as well as the material required for detecting precedences among the tasks and adjusting earliest starting times.

Robust Edge-Finding rule for filtering the earliest starting times

Let $\Theta^0 \subseteq \mathcal{I}^0, \Theta^1 \subseteq \mathcal{I}^1$ and $i^b \in (\mathcal{I}^0 \cup \mathcal{I}^1) \setminus (\Theta^0 \cup \Theta^1)$ be such that $b \in \{0,1\}, |\Theta^1| \leq r-b$ and $\mathcal{I}(\Theta^0) \cap \mathcal{I}(\Theta^1) = \emptyset$. Then the following states the Edge-Finding rule

$$\mathcal{C}(\max(\mathrm{lct}_{\mathcal{I}(\Theta^0)}, \mathrm{lct}^d_{\mathcal{I}(\Theta^1)}) - \mathrm{est}_{\mathcal{I}(\Theta^0 \cup \Theta^1 \cup \{i^b\})}) < \mathrm{e}_{\Theta^0} + \mathrm{e}_{\Theta^1} + \mathrm{e}_{i^b} \Rightarrow \Theta^0 \cup \Theta^1 \prec i^b \tag{5.15}$$

Equation (5.15) is implied from the fact that if the execution of i^b along with a set $\Theta^0 \cup \Theta^1$ with at most $r-b$ delayed tasks causes a failure due to the Overload Checking, then i^b should complete after $\Theta^0 \cup \Theta^1$.

Example 5.3.1. Let $\mathcal{I} = \{A, B, C, D\}$ be the set of tasks in table 5.2 which must execute on a resource of capacity $\mathcal{C} = 7$ in the context where at most $r = 1$ tasks are allowed to delay. According to (5.15), the precedence $\{B^0, D^0\} \prec C^1$ is deduced for $\Theta^0 = \{B^0, D^0\}, \Theta^1 = \emptyset, i = C$ and $b = 1$, as 154 = 7(28 - 6) < 63 + 72 + 36 = 171 and the precedence $\{A^0, B^1\} \prec D^0$ holds for $\Theta^0 = \{A^0\}, \Theta^1 = \{B^1\}, i = D$ and $b = 0$, as 161 = 7(24 - 1) < 24 + 70 + 72 = 166.

task	est	lct	p	d	c
A	1	13	4	1	6
B	6	23	9	1	7
C	16	36	8	10	2
D	14	28	12	1	6

Table 5.2 – A set of tasks $\mathcal{I} = \{A, B, C, D\}$ to execute on a resource of capacity $\mathcal{C} = 7$.

Robust $\Lambda-$earliest energy envelope

Similar to the discussion in section 3.3.1, the Edge-Finding in the robust context can also be implemented such that during the processing of the tasks, the regular or delayed tasks which make the precedences are maintained in a subset of tasks $\Lambda \subset \mathcal{I}$. Initially the tasks belong to Θ^0 and Θ^1 and the idea is to check whether adding one task from Λ^0 to Θ^0 or adding one task from Λ^1 to Θ^1 leads to $\text{Env}(\Theta^0, \Theta^1) > \mathcal{C} \cdot \max(\text{lct}_{\mathcal{I}(\Theta^0)}, \text{lct}^d_{\mathcal{I}(\Theta^1)})$. As soon as such a task is found in Λ, the established precedence is recorded and the task gets unscheduled from Λ. Assuming that Λ^0 and Λ^1 respectively contain the regular and delayed tasks from Λ, a variant of the earliest energy envelope of the tasks in $\Theta \cup \Lambda$, when one task from Λ is selected and at most r tasks are delayed, is defined as follows.

$$\text{Env}^{\Lambda r}(\Theta^0, \Theta^1, \Lambda^0, \Lambda^1) = \max(\max_{i^0 \in \Lambda^0} \text{Env}^r(\Theta^0 \cup \{i^0\}, \Theta^1), \max_{i^1 \in \Lambda^1} \text{Env}^r(\Theta^0, \Theta^1 \cup \{i^1\})) \quad (5.16)$$

$\text{Env}^{\Lambda r}(\Theta^0, \Theta^1, \Lambda^0, \Lambda^1)$ is the largest envelope that can be taken by taking a task from Λ^0 or Λ^1 and adding to Θ^0 and Θ^1. In the following, we present a data structure from which $\text{Env}^{\Lambda r}(\Theta^0, \Theta^1, \Lambda^0, \Lambda^1)$ can be retrieved efficiently.

$(\Theta - \Lambda)^r_L-$tree

Recall from section 3.3.1 that Vilím extends the $\Theta-$tree to the $(\Theta - \Lambda)-$tree which is primarily different from $\Theta-$tree in that it keeps track of the tasks in Λ for which a precedence can exist. This is a feature that is not required to be addressed in the $\Theta-$tree. Furthermore, the nodes of $(\Theta - \Lambda)-$tree maintain additional parameters for the energy and envelope of the tasks which belong to Λ. Analogous to the notion of Θ^r_L-tree, we define the $(\Theta - \Lambda)^r_L-$tree which is an extension of $(\Theta - \Lambda)-$tree. In addition to the parameters e^k_v and Env^k_v as in (5.9) and (5.10), this tree maintains the parameters $\Lambda-$energy, denoted $\text{e}^{\Lambda k}_v$, and $\Lambda-$earliest energy envelope, denoted $\text{Env}^{\Lambda k}_v$, associated to the tasks in Λ as follows.

$$\text{e}^{\Lambda k}_v = \begin{cases} c_v p_v & \text{if } (v^0 \in \Lambda^0) \wedge (k = 0 \ \vee \ v^1 \notin \Lambda^1) \\ c_v p^d_v & \text{if } (v^1 \in \Lambda^1) \wedge (k > 0) \\ -\infty & \text{otherwise} \end{cases} \quad (5.17)$$

$$\text{Env}^{\Lambda k}_v = \begin{cases} \mathcal{C}\, \text{est}_v + \text{e}^{\Lambda k}_v & \text{if } (v^0 \in \Lambda^0) \cup (v^1 \in \Lambda^1) \\ -\infty & \text{otherwise} \end{cases} \quad (5.18)$$

Similar to the Θ_L^r-tree, the superscript k stands for an upper bound on the number of delayed tasks in the leaf v. In order to make the computations of the $\Lambda-$energy and $\Lambda-$earliest energy envelope symmetrical over all nodes of the tree, we suppose that $0 \leq k \leq r$ and since $\mathrm{e}_v^{\Lambda k}$ and $\mathrm{Env}_v^{\Lambda k}$ are the $\Lambda-$energy and $\Lambda-$earliest energy envelope when at most k tasks are delayed, for $k \geq 2$ in the leaves of the $(\Theta - \Lambda)_L^r-$tree we necessarily have $\mathrm{e}_v^{\Lambda k} = \mathrm{e}_v^{\Lambda 1}$ and $\mathrm{Env}_v^{\Lambda k} = \mathrm{Env}_v^{\Lambda 1}$.

Let w be an internal node of the tree. $\mathrm{e}_w^{\Lambda k}$ and $\mathrm{Env}_w^{\Lambda k}$ are the maximum $\Lambda-$energy and the $\Lambda-$earliest energy envelope of the tasks in Θ whose leaves are descendant of w and to which one task from Λ is added. The task from Λ is also a descendant of w. According to the formulae mentioned in section 3.3.1, we have

$$\mathrm{e}_w^{\Lambda 0} = \max(\mathrm{e}_{\text{left}(w)}^{0} + \mathrm{e}_{\text{right}(w)}^{\Lambda 0}, \mathrm{e}_{\text{left}(w)}^{\Lambda 0} + \mathrm{e}_{\text{right}(w)}^{0}) \tag{5.19}$$

$$\mathrm{Env}_w^{\Lambda 0} = \max(\mathrm{Env}_{\text{left}(w)}^{\Lambda 0} + \mathrm{e}_{\text{right}(w)}^{0}, \mathrm{Env}_{\text{right}(w)}^{\Lambda 0}, \mathrm{Env}_{\text{left}(w)}^{0} + \mathrm{e}_{\text{right}(w)}^{\Lambda(0)}) \tag{5.20}$$

When scheduling, the regular tasks i^0 are added to Λ^0 and the delayed tasks i^1 are added to Λ^1. When unscheduling, the regular tasks i^0 are removed from Λ^0 and the delayed tasks i^1 are removed from Λ^1. Then, the nodes corresponding to the task are updated according to (5.17) and (5.18).

At most k tasks are delayed and in the computation of $\mathrm{e}_w^{\Lambda k}$ one task from Λ contributes. This task could be among at most j delayed tasks in the left subtree or at most $k - j$ tasks in the right subtree emanating from the inner node w, $0 \leq k \leq r, 0 \leq j \leq k$. According to (5.19) and (5.20)

$$\mathrm{e}_w^{\Lambda k} = \max_{0 \leq j \leq k} \{\mathrm{e}_{\text{left}(w)}^{j} + \mathrm{e}_{\text{right}(w)}^{\Lambda(k-j)}, \mathrm{e}_{\text{left}(w)}^{\Lambda j} + \mathrm{e}_{\text{right}(w)}^{k-j}\} \tag{5.21}$$

$$\mathrm{Env}_w^{\Lambda k} = \max_{0 \leq j \leq k} \{\mathrm{Env}_{\text{left}(w)}^{\Lambda j} + \mathrm{e}_{\text{right}(w)}^{k-j}, \mathrm{Env}_{\text{left}(w)}^{j} + \mathrm{e}_{\text{right}(w)}^{\Lambda(k-j)}\} \cup \{\mathrm{Env}_{\text{right}(w)}^{\Lambda k}, \mathrm{Env}_w^{\Lambda(k-1)}\} \tag{5.22}$$

In the case that the number of delayed task is greater than the number of available nodes in the subtree of the right side, it is sufficient to retrieve $\mathrm{Env}_w^{\Lambda(k-1)}$ in (5.22), i.e. the lambda energy envelope when at most $k - 1$ tasks are delayed. Finally, if w is the root of $(\Theta - \Lambda)_L^r-$tree, the function $\mathrm{Env}^{\Lambda r}(\Theta^0, \Theta^1, \Lambda^0, \Lambda^1)$ can be computed by computing the value $\mathrm{Env}_{\text{root}}^{\Lambda k}$ at the root node.

Robust Edge-Finding algorithm for filtering the earliest starting times

The implementation of Edge-Finding proceeds in two phases. Firstly, the existing precedences among the tasks are detected. Thereafter, the earliest starting time of the tasks subject to a precedence are adjusted.

Detection phase

Algorithm 11 adapts the detection phase of the Edge-Finding for the earliest starting times. This algorithm emulates the algorithm that Vilím [84] proposes. The algorithm starts with a full $(\Theta -$

$\Lambda)^r_L-$tree, in which all the regular as well as the delayed tasks are scheduled in $\Theta = \Theta^0 \cup \Theta^1$ and $\Lambda = \Lambda^0 \cup \Lambda^1$ is empty. That is,

$$\Theta^0 = \mathcal{I}^0, \Theta^1 = \mathcal{I}^1, \Lambda^0 = \Lambda^1 = \emptyset$$

The algorithm iterates over the set of all latest completion times and delayed latest completion times $T = \{\mathrm{lct}_i : i \in \mathcal{I}\} \cup \{\mathrm{lct}^d_i : i \in \mathcal{I}\}$ in non-increasing order. First, the algorithm makes sure that the Overload Checking does not fail (line 10). If so, for every $t \in T$

$$\mathrm{Env}^{\Lambda r}(\Theta^0, \Theta^1, \Lambda^0, \Lambda^1) > \mathcal{C} \cdot t \tag{5.23}$$

is checked which captures the precedence. Thanks to the structure of the $(\Theta-\Lambda)^r_L-$tree, $\mathrm{Env}^{\Lambda r}(\Theta^0, \Theta^1, \Lambda^0, \Lambda^1)$ is retrieved from the root of $(\Theta - \Lambda)^r_L-$tree for each t. Line 13 retrieves the task subject to a precedence. It can be implemented in $O(\log(n))$ time by traversing down the tree. The algorithm proceeds by traversing down the $(\Theta - \Lambda)^r_L-$tree from the root. At each inner node w in such a traversal, the algorithm determines which one of the cases in (5.22) satisfy for $\mathrm{Env}^{\Lambda k}_w$. So long as this task is taken from $\mathrm{Env}^{\Lambda k}_f$, where f is a child of w, the algorithm continues down. As soon as the task is taken from an $\mathrm{e}^{\Lambda k}_f$, the algorithm switches to check the cases of (5.21). The task subject to a precedence is a task $i^b \in \Lambda^b, b \in \{0, 1\}$, and only one task in $\Lambda^0 \cup \Lambda^1$ is used to compute $\mathrm{Env}^{\Lambda k}_w$. The candidate task with such a property is called the *responsible task*. The responsible task, which is located on a leaf of the $(\Theta - \Lambda)^r_L-$tree, causes (5.16) to be maximized, hence making a precedence. Contingent upon the delay status of i, we encode the precedence that i creates in a two dimensional matrix by 0 and 1 columns, indicating whether the task is regular or delayed (line 14). The precedences detected during the detection phase are encoded in the *prec* array. For $b \in \{0, 1\}$, $\mathrm{prec}[i, b] = t$ means that i^b is preceded by the subsets of tasks from $\Theta^0 \cup \Theta^1$ with at most $r - b$ delayed tasks which terminate no later than t and cause (5.16) to be maximized. Once the precedence is recorded on line 14, the algorithm unschedules i^b from Λ^b. After the execution of the loop at line 12, all the tasks whose delayed latest completion time equals t are unscheduled from Θ^1 and they are rather scheduled in Λ^1. Furthermore, all the tasks whose latest completion time equals t, are removed from Θ^0 and Θ^1 and scheduled in Λ^0

and Λ^1.

Algorithm 11: DetectionPhaseOfEdge-FindingForLowerBounds($\mathcal{I}$, $\mathcal{C}$, r)

```
1  T ← {lct_l : l ∈ I} ∪ {lct_l^d : l ∈ I}
2  for i ∈ {1, ..., n} do
3      prec[i, 0] ← −∞
4      prec[i, 1] ← −∞
5  Θ^0 ← I^0
6  Θ^1 ← I^1
7  Λ^0 ← ∅
8  Λ^1 ← ∅
9  for t ∈ T in non-increasing order do
10     if Env^r(Θ^0, Θ^1) > C · t then
11         fail
12     while Env_root^{Λr} > C · t do
13         i^b ← The task in Λ^0 ∪ Λ^1 that maximizes Env^{Λr}(Θ^0, Θ^1, Λ^0, Λ^1)
14         prec[i, b] ← t
15         Λ^b ← Λ^b \ {i^b}
16     Δ^1 = {i^1 ∈ I^1 | lct_i^d = t}
17     Δ^0 = {i^0 ∈ I^0 | lct_i = t}
18     Λ^1 ← Λ^1 ∪ Δ^1
19     Θ^1 ← Θ^1 \ Δ^1
20     Λ^0 ← Λ^0 ∪ Δ^0
21     Θ^0 ← Θ^0 \ Δ^0
```

In order to elucidate the mechanism of the algorithm (11), in figure 5.2 we present the first few steps of the detection phase for the tasks of example 5.3.1. For this example, $T = \{46, 36, 29, 28, 24, 23, 14, 13\}$. Figure 5.2a illustrates the initialization step of the algorithm, where the $(\Theta - \Lambda)^r_L$−tree is full. That is, all the regular as well as delayed tasks are scheduled and no task is a candidate yet to create a precedence. In the first iteration $t = 46$ is processed, which corresponds to $\text{lct}^d_C = t$. C^1 gets unscheduled from Θ^1 and rather scheduled in Λ^1, to be flagged as a delayed tasks which could create a precedence later. The updated status of the node corresponding to C^1 is depicted in figure 5.2b. In the second iteration, $t = 36$ is processed, which corresponds to $\text{lct}_C = t$. At this point, $\text{Env}^{\Lambda r}_{\text{root}} = 213 \not> \mathcal{C} \cdot t$, which implies that (5.16) is not great enough to detect a precedence. Therefore, the loop dedicated to detect an existing precedence fails to execute and C^0 is removed from Θ^0 and fully scheduled in Λ. Figure 5.2c corresponds to this case after updating the entire tree. The next iteration processes $t = \text{lct}^d_D = 29$. Since $\text{Env}^{\Lambda r}_{\text{root}}$ =213 > $\mathcal{C} \cdot t$, the while loop executes. A traversal down the tree to find the responsible task locates C^1. Once the precedence $\text{prec}[C^1, 1] = 29$ is recorded, C^1 gets unscheduled from Λ^1, as illustrated in figure 5.2d. For this iteration there is no more precedences

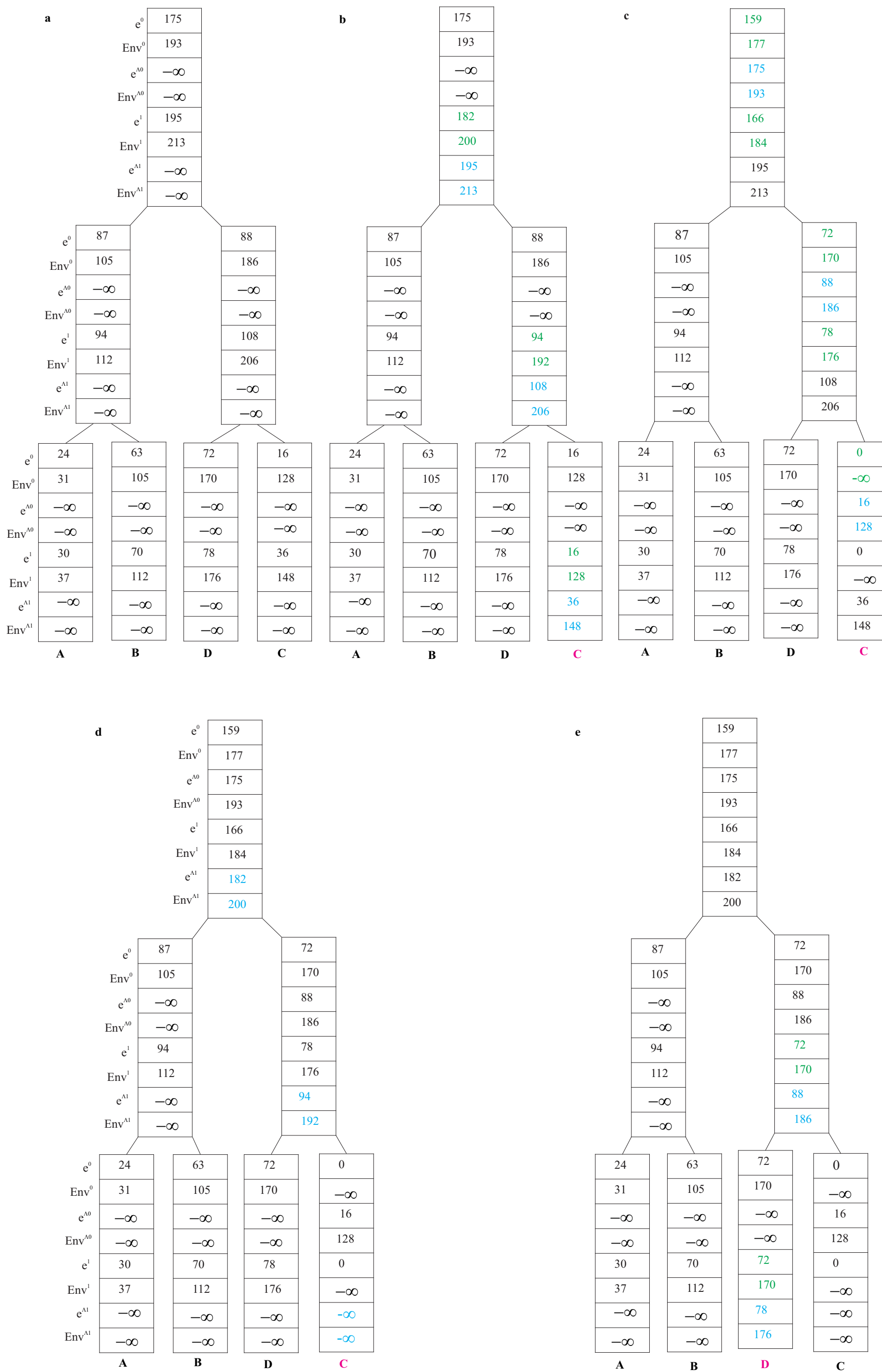


Figure 5.2 – In the picture (a), the algorithm starts with a full $(\Theta - \Lambda)^r_L-$tree. In the picture (b), C^1 gets unscheduled from Θ^1 and scheduled in Λ^1. The modifications to Θ and Λ sets are respectively coloured in green and blue. In the picture (c), C^0 is unscheduled from Θ^0 and scheduled in Λ^0. In the picture (d) after C^1 is found as the responsible tasks, it gets unscheduled from Λ^1. In the picture (e), D^1 gets unscheduled from Θ^1 and scheduled in Λ^1.

Task \ b	0	1
A	$-\infty$	$-\infty$
B	14	14
C	28	29
D	24	24

Table 5.3 – The *prec* array which is obtained after the execution of the algorithm 11 on the instance of example 5.3.1

Algorithm 12: Maxest(tree,bound,c,$\mathcal{C}$,k,o)

```
1  v ← root
2  e[0...k] ← 0⃗
3  maxEnv^c ← (C − c) · bound
4  k ← k − o
5  while v is not a leaf do
6      branchRight ← false
7      for j = 0, ..., k do
8          if Env^{c,j}_{right(v)} + e[k − j] > maxEnv^c then
9              v ← right(v)
10             branchRight ← true
11             break
12     if not branchRight then
13         v ← left(v)
14         for j = 0, ..., k do
15             e′[j] ← max_{0≤m≤j} e[j] + e^{i−j}_{right(v)}
16         e ← e′
17 return v
```

to detect. Thus, the while loop terminates and D^1 for which $\text{lct}_D^d = 29$ gets unscheduled from Θ^1 and scheduled in Λ^1 as depicted in figure 5.2e. In further executions of the algorithm, the precedence $\{A^0, B^1\} \prec D^0$ will be detected, as well. Table 5.3 indicates the *prec* array as the result of the algorithm 11.

Adjustment phase

The adjustment of the earliest starting times is done by iterating over the detected precedences which are recorded in the prec array. Let i^b be the tasks for which a precedence was detected with (5.15) and and $\text{prec}[i, b] = t$. For $\Omega^0 \subseteq \Theta^0, \Omega^1 \subseteq \Theta^1, \emptyset \neq \Omega^0 \cup \Omega^1$ such that $|\Omega^1| \leq r - b$, we define a variation of (3.12) in the robust context as:

$$\text{rest}(\Omega^0, \Omega^1, c_i) = \text{e}_{\Omega^0 \cup \Omega^1} - (\mathcal{C} - c_i)(\max(\text{lct}_{\mathcal{I}(\Omega^0)}, \text{lct}^d_{\mathcal{I}(\Omega^1)}) - \text{est}_{\Omega^0 \cup \Omega^1}) \tag{5.24}$$

The following formula adjusts the earliest starting time of i.

$$\text{est}_i \leftarrow \max(\text{est}_i, \max_{\substack{\Omega^0 \subseteq \text{Lcut}^0(t) \setminus \{i^0\} \cap \{j^0 \in \mathcal{I}^0 : \text{lct}_j \le t < \text{lct}_j^d\} \\ \Omega^1 \subseteq \text{Lcut}^1(t) \setminus \{i^1\} \\ \emptyset \neq \Omega^0 \cup \Omega^1 \\ |\Omega^1| \le r - b \\ \text{rest}(\Omega^0, \Omega^1, c_i) > 0}} \{\text{est}_{\Omega^0 \cup \Omega^1} + \left\lceil \frac{\text{rest}(\Omega^0, \Omega^1, c_i)}{c_i} \right\rceil \}) \tag{5.25}$$

From the condition $\text{rest}(\Omega^0, \Omega^1, c_i) > 0$ we obtain

$$(\mathcal{C} - c_i)(\text{est}_{\Omega^0 \cup \Omega^1}) + \text{e}_{\Omega^0 \cup \Omega^1} > (\mathcal{C} - c_i)(\max(\text{lct}_{\mathcal{I}(\Omega^0)}, \text{lct}^d_{\mathcal{I}(\Omega^1)})) \tag{5.26}$$

The left side of (5.26) describes an equivalent of $\text{Env}^r(\Theta^0, \Theta^1)$, defined by (5.8), on a resource with capacity $\mathcal{C} - c_i$. This motivates the idea of defining a variant of the earliest energy envelope, defined as (3.2), with respect to the capacity c of a task [84] as

$$\text{Env}^c_\Theta = \max_{\Omega \subseteq \Theta}((\mathcal{C} - c)\,\text{est}_\Omega + \text{e}_\Omega) \tag{5.27}$$

and a variant of (5.8) as

$$\text{Env}^{cr}(\Theta^0, \Theta^1) = \max_{\substack{\Omega^0 \subseteq \Theta^0 \\ \Omega^1 \subseteq \Theta^1 \\ |\Omega^1| \le r \\ \mathcal{I}(\Omega^0) \cap \mathcal{I}(\Omega^1) = \emptyset}}((\mathcal{C} - c)\,\text{est}_{\mathcal{I}(\Omega^0 \cup \Omega^1)} + \text{e}_{\Omega^0} + \text{e}_{\Omega^1}) \tag{5.28}$$

The algorithm 14 indicates our adaption of the adjustment phase of the Edge-Finding for this purpose. For the rest of this section, we explain how this algorithm emulates the algorithm that Vilím proposes [84].

Let F be the set of all distinct capacities. In [84] the adjustment is done by processing all $c \in F$ in an outer loop and initializing a Θ^c-tree for each $c \in F$ in an inner loop. Θ^c-tree is an extension of Θ-tree in that in addition to maintaining the parameters energy and earliest energy envelope, every inner node holds the additional parameter (5.27). Analogous to the parameters of energy and earliest energy envelope in the $\Theta-$tree, for a leaf v of the Θ^c-tree, which corresponds to a task $i \in \mathcal{I}$, (5.27) is computed for $\Theta = \{i\}$ and for an inner node w of the Θ^c-tree, (5.27) is computed for the set of all tasks which are included in the subtree rooted at w. This computation is done recursively as below.

$$\text{Env}^c_w = \max(\text{Env}^c_{\text{left}(w)} + \text{e}^0_{\text{right}(w)}, \text{Env}^c_{\text{right}(w)}) \tag{5.29}$$

Note that in [84] all distinct capacities $c \in F$ are considered individually, no matter a precedence was detected per capacity or not. For the sake of efficiency, we rather consider F as the set of capacities of the tasks for which a precedence was detected. For each $c \in F$, we initialize an empty Θ^{cr}_L-tree,

which is an extension of Θ_L^r-tree. With regard to the material presented in section 5.2.4, a leaf v of the $\Theta_L^{cr}-$tree in addition to (5.9) and (5.10) holds

$$\mathrm{Env}_v^{ck} = \begin{cases} (\mathcal{C} - c)\,\mathrm{est}_v + \mathrm{e}_v^k & \text{if } (v^0 \in \Theta^0) \vee (v^1 \in \Theta^1) \\ -\infty & \text{otherwise} \end{cases} \tag{5.30}$$

and Env_w^{ck} for an internal node w is computed as

$$\mathrm{Env}_w^{ck} = \max_{0 \le j \le k} \{\mathrm{Env}_{\text{left}(w)}^{cj} + \mathrm{e}_{\text{right}(w)}^{k-j}\} \cup \{\mathrm{Env}_{\text{right}(w)}^{ck}, \mathrm{Env}_w^{c(k-1)})\} \tag{5.31}$$

Θ^c-tree develops by processing all lct_j for $j \in \mathcal{I}$ in non-decreasing order and scheduling j by adding it to Θ. Rather than iterating through all lct_j and lct_j^d for $j \in \mathcal{I}$, we develop the $\Theta_L^{cr}-$tree by iterating over the tasks i^b in non-decreasing order of prec$[i, b]$. The adjustment should be done only if $c_i = c$. If so, at iterating each prec$[i, b] = t$, all the regular or delayed tasks which cannot complete after t are scheduled in the $\Theta_L^{cr}-$tree by adding to Θ^0 or Θ^1.

Let $\Omega^0 \cup \Omega^1$ be the candidate subset which can give the strongest adjustment in (5.25). Once all the tasks which are eligible to be in $\Omega^0 \cup \Omega^1$ are scheduled in the tree, it is time to identify Ω^0 and Ω^1 and compute the value of the second component of the right side of (5.25). Emulating [84], we proceed to compute Ω^0 and Ω^1 in two steps.

The first step is to compute the maximum earliest starting time (or *maxest*) for $\Omega^0 \cup \Omega^1$. There might be multiple sets $\Omega^{0\prime}$ and $\Omega^{1\prime}$ that satisfy (5.25) and the goal is to compute the largest $\mathrm{est}_{\Omega^{0\prime} \cup \Omega^{1\prime}}$ for the appropriate $\Omega^{0\prime}$ and $\Omega^{1\prime}$. The new variant of the earliest energy envelope, defined in (5.27), helps to locate the node responsible for maxest. In the algorithms 12, we consider this idea for identifying maxest for $\Omega^0 \cup \Omega^1$ in (5.25). Note that line 11 of algorithm 14 retrieves the largest completion time for the set of all tasks that are scheduled so far, whether they are regular or delayed. This value, named *bound*, which is given as a parameter to the algorithm 12, is important for checking the right side of (5.26) in this algorithm.

The second step is to compute the envelope of the tasks which is done in the algorithm 13 . This is done by dividing the tasks in two groups. The tasks which start before or at maxest and the rest of the tasks which start after maxest. The tasks which qualify for the former case belong to a set $\alpha(j, c)$ and the tasks which qualify for the latter case belong to a set $\beta(j, c)$. From the sets $\alpha(j, c)$ and $\beta(j, c)$, the earliest energy envelop in the $\Theta_L^{cr}-$tree is computed with

$$\mathrm{Env}(j, c) = \mathrm{e}_\beta + \mathrm{Env}_{\alpha(j,c)} \tag{5.32}$$

where e_β and Env_α are computed in a bottom-up manner, starting from the located task responsible for maxest, and by taking into account that at most r tasks are delayed. Once the envelope is computed, the value which makes the strongest update is adjusted by taking the maximum between the envelope computed in the current iteration and the preceding iterations (line 15). Ultimately, if the task can not finish after t, it gets scheduled itself.

Here is the adjustment of the precedence $\{B^0, D^0\} \prec C^1$ for the task C from the example 5.3.1. Figure 5.3 illustrates the $\Theta_L^{cr}-$tree for the state where est_C is adjusted. The adjustment is done when processing $c = 2$. At this state, $\Theta^0 = \{A^0, B^0, D^0\}$ and $\Theta^1 = \{A^1, B^1\}$. The set of scheduled tasks has, for largest completion completion time, $m = 28$ in line 11 of the algorithm. The blue arrows in figure 5.3 show the path which is traversed by the algorithm 12. This algorithm locates the yellow node which corresponds to the task D for maxest. The algorithm 13 starts from the yellow node and the green arrows show the path which is actually the reversed blue path traversed by this algorithm to the root. This algorithm computes $\text{E}nv = 184$ for line 13 and in line 14 the difference is computed diff = $\lceil 184 - (7-2)28/2 \rceil = 22$. Hence, est_C gets filtered to $\text{est}_C = 22$. Proceeding the adjustment phase for the precedence $\{A^0, B^1\} \prec D^0$ in a similar fashion yields $\text{est}_D = 15$.

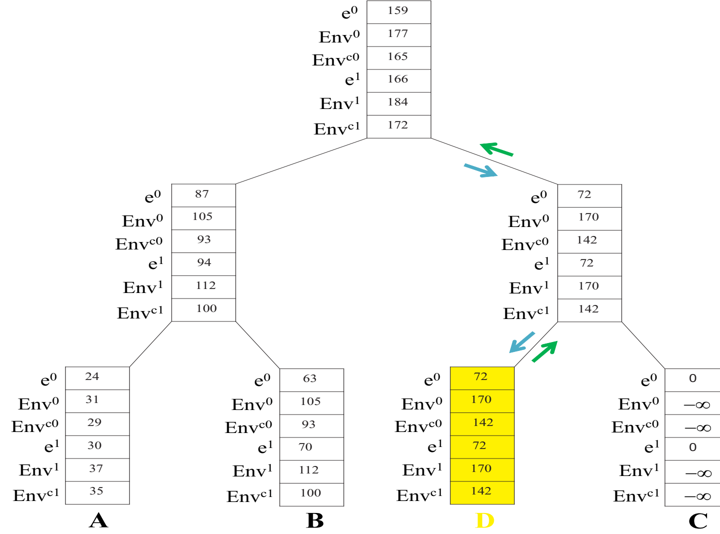


Figure 5.3 – The state of $\Theta_L^{cr}-$tree in the iteration that est_C gets adjusted. The blue arrows show the path which is traversed by the algorithm 12. This algorithm locates the yellow node. The algorithm 13 starts from the yellow node and the green arrows show the path traversed by this algorithm to the root.

Lemma 5. The time complexity of the robust Edge-Finding is $O(r^2 kn \log(n))$.

Proof. Unscheduling a task from Θ^0 or Θ^1 or scheduling in Λ^0 or Λ^1 requires to update the values e_i and Env_i, $\text{e}_v^{\Lambda j}$ and $\text{Env}_v^{\Lambda j}$ of the leaf of the tree as well as all nodes up to the root of the tree. Therefore, the lines 18, 19, 20 and 21 of the algorithm 11 run in $O(r^2 \log(n))$. This complexity is maintained for finding the responsible task, finding the maxest as well as computing the envelope, since for such operations the tree is traversed from the root to the leaves or conversely. The scheduling and unscheduling tasks at lines 9 and 10 of the algorithm 14 occur at most n times, each time implying in $O(r^2 \log(n))$ computations. Moreover, since each task is unscheduled once from Θ^0 and once from Θ^1 or scheduled once in Λ^0 and once in Λ^1, considering that there are k distinct capacities, the overall time complexity is $O(r^2 kn \log(n))$. □

Algorithm 13: EnvelopeForLowerBound(v,tree,k)

```
1  e_α ← [e_v^0, e_v^1, 0, ..., 0] // e_α includes k - 1 entries 0
2  e_β ← 0⃗
3  Env_α ← [Env_v^0, Env_v^1, −∞, ..., −∞] // Env_α includes k - 1 entries −∞
4  while v is not the root do
5      if v is a left child then
6          e'_β ← 0⃗
7          for j = 0 to k do
8              e'_β[j] ← max_{0≤i≤j}(e_β[i] + e_{sibling(v)}^{j−i}, e'_β[j − 1])
9          e_β ← e'_β
10     else
11         Env'_α ← [−∞, ..., −∞]
12         e'_α ← 0⃗
13         for j = 0, ..., k do
14             Env'_α[j] ← max(max_{0≤i≤j}(Env_{sibling(v)}^i + e_α[j − i]), Env_α[j], Env'_α[j − 1])
15             e'_α[j] ← max(max_{0≤i≤j}(e_{sibling(v)}^i + e_α[j − i]), e'_α[j − 1])
16     Env_α ← Env'_α
17     e_α ← e'_α
18     v ← parent(v)
19 return max_{0≤i≤k}(e_β[i] + Env_α[k − i])
```

5.3.2 Filtering the latest completion times

This section discusses the Edge-Finding rule, as well as the material required for detecting precedences among the tasks and adjusting latest completion times.

Robust Edge-Finding rules for filtering the latest completion times

Let $\Theta^0 \subseteq \mathcal{I}^0, \Theta^1 \subseteq \mathcal{I}^1$ and $i^0 \in \mathcal{I}^0 \setminus \Theta^0$ be such that $\mathcal{I}(\Theta^0) \cap \mathcal{I}(\Theta^1) = \emptyset$. The following states the first Edge-Finding rule

$$\mathcal{C}(\max(\text{lct}_{\mathcal{I}(\Theta^0)\cup\{i\}}, \text{lct}^d_{\mathcal{I}(\Theta^1)}) - \text{est}_{\mathcal{I}(\Theta^0\cup\Theta^1)}) < \text{e}_{\Theta^0} + \text{e}_{\Theta^1} + \text{e}_{i^0} \Rightarrow i^0 \prec \Theta^0 \cup \Theta^1 \tag{5.33}$$

If $i^1 \in \mathcal{I}^1 \setminus \Theta^1$ be such that $\mathcal{I}(\Theta^0) \cap \mathcal{I}(\Theta^1) = \emptyset$, then the following states the second Edge-Finding rule

$$\mathcal{C}(\max(\text{lct}_{\mathcal{I}(\Theta^0)}, \text{lct}^d_{\mathcal{I}(\Theta^1)\cup\{i\}}) - \text{est}_{\mathcal{I}(\Theta^0\cup\Theta^1)}) < \text{e}_{\Theta^0} + \text{e}_{\Theta^1} + \text{e}_{i^1} \Rightarrow i^1 \prec \Theta^0 \cup \Theta^1 \tag{5.34}$$

Example 5.3.2. After adjusting est_C and est_D due to the precedences detected for the tasks C and D in example 5.3.1, the tasks are updated as follows:

Algorithm 14: AdjustmentOfLowerBounds(prec, r, $\mathcal{C}$)

1 $F \leftarrow$ The capacities of the tasks for which a precedence was detected
2 **for** $c \in F$ **do**
3 $\quad \Theta^0 \leftarrow \emptyset$
4 $\quad \Theta^1 \leftarrow \emptyset$
5 $\quad$ upd $\leftarrow [-\infty, -\infty]$
6 $\quad t' \leftarrow 0$
7 $\quad$ **for** $i^b \in \{j^b \mid prec[j,b] > -\infty \wedge c_i = c\}$ *in non-deccreasing order of prec[i, b]* **do**
8 $\quad\quad t \leftarrow \text{prec}[i,b]$
9 $\quad\quad \Theta^1 \leftarrow \Theta^1 \cup (\text{Lcut}^1(t) \setminus \text{Lcut}^1(t')) \setminus \{i^1\}$
10 $\quad\quad \Theta^0 \leftarrow \Theta^0 \cup \{j^0 \in \mathcal{I}^0 : \text{lct}_j \leq t < \text{lct}_j^d\} \setminus \{j^0 \in \mathcal{I}^0 : \text{lct}_j \leq t' < \text{lct}_j^d\}) \setminus \{i^0\}$
11 $\quad\quad m \leftarrow \max(\{\text{lct}_j \mid j^0 \in \Theta^0\} \cup \{\text{lct}_j^d \mid j^1 \in \Theta^1\})$
12 $\quad\quad v \leftarrow \text{Maxest}(\Theta, m, c, \mathcal{C}, r, b)$
13 $\quad\quad$ Env $\leftarrow$ EnvelopeForLowerBound$(v, \Theta, r - b)$
14 $\quad\quad$ diff $\leftarrow \lceil (\text{E}nv - (\mathcal{C} - c)m/c \rceil$
15 $\quad\quad$ upd[b] $\leftarrow$ max(upd[b],diff)
16 $\quad\quad \text{est}_i \leftarrow \max(\text{est}_i,$ upd[b] $)$
17 $\quad\quad$ **if** $\text{lct}_i \leq t \wedge \text{lct}_i^d \leq t$ **then** $\Theta^1 \leftarrow \Theta^1 \cup \{i^1\}$
18 $\quad\quad$ **else if** $\text{lct}_i \leq t$ **then** $\Theta^0 \leftarrow \Theta^0 \cup \{i^0\}$
19 $\quad\quad t' \leftarrow t$

task	est	lct	p	d	c
A	1	13	4	1	6
B	6	23	9	1	7
C	22	36	8	10	2
D	15	28	12	1	6

This update causes two new precedences to be detected by filtering the latest completion times. Assuming $\Theta^0 = \{B^0\}, \Theta^1 = \{D^1\}, i = A, b = 0$, the precedence $A^0 \prec \{D^1, B^0\}$ holds as 161 = 7(29 - 6) < 24 + 63 + 78 = 165 and assuming $\Theta^0 = \{B^0\}, i = D, b = 0$ the precedence $B^0 \prec D^0$ holds as 91 = 7(28 - 15) < 63 +72 = 135.

Robust latest energy envelope

One can generalize the notion of the latest energy envelope, defined with (3.3), for the case that the tasks can be delayed. Rather than computing the latest energy envelope of a set Θ as in (3.3), we compute the latest energy envelope of two sets $\Theta^0 \subseteq \mathcal{I}^0$ and $\Theta^1 \subseteq \mathcal{I}^1$. The tasks in Θ^0 can be regular while the tasks in Θ^1 can be delayed. The tasks that belong to both sets can either be regular

or delayed but not both. The latest energy envelope for the sets Θ^0 and Θ^1 in such a case is defined

$$\mathrm{Env}'^{r}(\Theta^0,\Theta^1) = \min_{\substack{\Omega^0\subseteq\Theta^0\\ \Omega^1\subseteq\Theta^1\\ |\Omega^1|\leq r\\ \mathcal{I}(\Omega^0)\cap\mathcal{I}(\Omega^1)=\emptyset}} (\mathcal{C}\max(\mathrm{lct}_{\mathcal{I}(\Omega^0)}, \mathrm{lct}^d_{\mathcal{I}(\Omega^1)}) - \mathrm{e}_{\Omega^0} - \mathrm{e}_{\Omega^1}) \tag{5.35}$$

Robust $\Lambda-$latest energy envelope

Similar to filtering the earliest starting times, the Edge-Finding for filtering the latest completion times can also be implemented such that during the processing of the tasks, the regular or delayed tasks which make the precedences are maintained in a subset of tasks $\Lambda = \Lambda^0 \cup \Lambda^1 \subset \mathcal{I}$. Initially the tasks belong to Θ and the idea is to check whether adding one task from Λ^0 to Θ or adding one task from Λ^1 to Θ leads to $\mathrm{Env}'(\Theta^0,\Theta^1) < \mathcal{C}\cdot \mathrm{est}_{\mathcal{I}(\Theta^0\cup\Theta^1)}$ for $\Theta^0 = \{i^0 \mid i \in \Theta\}$ and $\Theta^1 = \{i^1 \mid i \in \Theta\}$. As soon as such a task is found in Λ, the established precedence is recorded and the task gets unscheduled from Λ. Assuming that Λ^0 and Λ^1 respectively contain the regular and delayed tasks from Λ, a variant of the latest energy envelope of the tasks in $\Theta \cup \Lambda$, when one task from Λ is selected and at most r tasks are delayed, is defined as follows.

$$\mathrm{Env}'^{\Lambda r}(\Theta^0,\Theta^1,\Lambda^0,\Lambda^1) = \min(\min_{i^0\in\Lambda^0} \mathrm{Env}'^{r}(\Theta^0\cup\{i^0\},\Theta^1), \min_{i^1\in\Lambda^1} \mathrm{Env}'^{r}(\Theta^0,\Theta^1\cup\{i^1\})) \tag{5.36}$$

$\mathrm{Env}'^{\Lambda r}(\Theta^0,\Theta^1,\Lambda^0,\Lambda^1)$ is the smallest envelope that can be taken by taking a responsible task from Λ^0 or Λ^1 and adding to Θ^0 and Θ^1. In the following, we present a data structure from which $\mathrm{Env}'^{\Lambda r}(\Theta^0,\Theta^1,\Lambda^0,\Lambda^1)$ can be retrieved efficiently.

$(\Theta - \Lambda)^r_U-$tree

Analogous to the notion of $(\Theta-\Lambda)^r_L-$tree, we define the $(\Theta-\Lambda)^r_U-$tree in order to develop the detection algorithm for filtering latest completion times. However, compared with the the preceding section, there is a major discrepancy in the way we construct the $(\Theta-\Lambda)^r_U-$tree. We associate each task i to two leaves. The regular task i^0 is associated to a leaf as usual but the delayed task i^1 is associated to another leaf that takes into account only the delayed part of the task. One can interpret it as a task with processing time d_i. Moreover, the values computed in each node of the tree are a function of three sets: $\Theta \subseteq \mathcal{I}$, $\Lambda^0 \subset \mathcal{I}^0$, and $\Lambda^1 \subset \mathcal{I}^1$. For instance, with such an interpretation, in the example 5.3.2, A^0 has $p_A \cdot c_A = 4\cdot 6 = 24$ units of energy, while i^1 has $d_A \cdot c_A = 1 \cdot 6 = 6$ units of energy.

Let $T = \{\mathrm{lct}_i : i \in \mathcal{I}\} \cup \{\mathrm{lct}^d_i : i \in \mathcal{I}\}$ be the set of all latest completion times and delayed latest completion times sorted in ascending order. We construct the $(\Theta-\Lambda)^r_U-$tree with $2n$ leaves. The tree is initialized with $2n$ tasks: n regular tasks and n delayed tasks. The leaves are sorted by lct for the regular tasks and lct^d for the delayed tasks. For an arbitrary leaf v, the energy and envelopes are defined as

$$\mathrm{e}^{k}_{v^0} = \begin{cases} c_v p_v & \text{if } k \geq 0 \\ 0 & \text{if } v^0 \notin \Theta \end{cases}$$

$$\mathrm{e}^{k}_{v^1} = \begin{cases} 0 & \text{if } k = 0 \\ c_v d_v & \text{if } k > 0 \\ 0 & \text{if } v^1 \notin \Theta \end{cases}$$

$$\mathrm{Env}'^{k}_{v^0} = \begin{cases} \mathcal{C}\,\mathrm{lct}_v - \mathrm{e}^{k}_{v^0} & \text{if } k \geq 0 \\ \infty & \text{if } v^0 \notin \Theta \end{cases}$$

$$\mathrm{Env}'^{k}_{v^1} = \begin{cases} \infty & \text{if } k = 0 \\ \mathcal{C}\,\mathrm{lct}^{d}_{v} - c_v d_v & \text{if } k > 0 \\ \infty & \text{if } v^1 \notin \Theta \end{cases}$$

$$\mathrm{e}^{\Lambda k}_{v^0} = \begin{cases} c_v p_v & \text{if } (k \geq 0) \wedge (v^0 \in \Lambda^0) \\ -\infty & \text{if } v^0 \notin \Lambda^0 \end{cases}$$

$$\mathrm{e}^{\Lambda k}_{v^1} = \begin{cases} 0 & \text{if } (k = 0) \wedge (v^1 \in \Lambda^1) \\ (p_v + d_v) c_v & \text{if } (k > 0) \wedge (v^1 \in \Lambda^1) \\ -\infty & \text{if } v^1 \notin \Lambda^1 \end{cases}$$

$$\mathrm{Env}'^{\Lambda k}_{v^0} = \begin{cases} \mathcal{C}\,\mathrm{lct}_v - c_v p_v & \text{if } (k \geq 0) \wedge (v^0 \in \Lambda^0) \\ \infty & \text{if } v^0 \notin \Lambda^0 \end{cases}$$

$$\mathrm{Env}'^{\Lambda k}_{v^1} = \begin{cases} \infty & \text{if } (k = 0) \wedge (v^1 \in \Lambda^1) \\ \mathcal{C}\,\mathrm{lct}^{d}_{v} - (p^{d}_{v} + d_v) c_v & \text{if } (k > 0) \wedge (v^1 \in \Lambda^1) \\ \infty & \text{if } v^1 \notin \Lambda^1 \end{cases}$$

Similar to the $(\Theta - \Lambda)^r_L-$tree, the superscript k stands for an upper bound on the number of delayed tasks in the leaf v. In order to make the computations of the $\Lambda-$energy and $\Lambda-$latest energy envelope symmetrical over all nodes of the tree, we suppose that $0 \leq k \leq r$ and since $\mathrm{e}^{\Lambda k}_{v}$ and $\mathrm{Env}'^{\Lambda k}_{v}$ are the $\Lambda-$energy and $\Lambda-$latest energy envelope when at most k tasks are delayed, for $k \geq 2$ in the leaves of the $(\Theta - \Lambda)^r_L-$tree we necessarily have $\mathrm{e}^{\Lambda k}_{v} = \mathrm{e}^{\Lambda 1}_{v}$ and $\mathrm{Env}'^{\Lambda k}_{v} = \mathrm{Env}'^{\Lambda 1}_{v}$.

Scheduling a regular or delayed task in $\Lambda^0 \cup \Lambda^1$ is equivalent to adding the task i^0 to Λ^0 or the task i^1 to Λ^1 and updating the node corresponding to the task according to the formulae above. Unscheduling a regular or delayed task from $\Lambda^0 \cup \Lambda^1$ is equivalent to removing the task i^0 from Λ^0 or the task i^1 from Λ^1 and updating the node corresponding to the task according to the formulae above. Unscheduling a task i from Θ removes both i^0 and i^1.

Let w be an internal node of the tree. Scott [73] proves that

$$\mathrm{Env'}^{0}_{w} = \min(\mathrm{Env}^{0}_{\text{right}(w)} - \mathrm{e}^{0}_{\text{left}(w)}, \mathrm{Env'}^{0}_{\text{left}(w)}) \tag{5.37}$$

$\mathrm{Env'}^{\Lambda k}_{w}$ is the minimum $\Lambda-$latest energy envelope of the tasks in Θ whose leaves are descendant of w and to which one task from $\Lambda^0 \cup \Lambda^1$ is added. The task from $\Lambda^0 \cup \Lambda^1$ is also a descendant of w. With a reasoning similar to (5.14) and (5.22), the values of $\mathrm{Env'}^{k}_{w}$ and $\mathrm{Env'}^{\Lambda k}_{w}$ for at most k delayed tasks, $0 \leq k \leq r$, are recursively computed as

$$\mathrm{Env'}^{k}_{w} = \min_{0 \leq j \leq k} \{\mathrm{Env'}^{j}_{\text{right}(w)} - \mathrm{e}^{k-j}_{\text{left}(w)}\} \cup \{\mathrm{Env'}^{k}_{\text{left}(w)}, \mathrm{Env'}^{(k-1)}_{w}\} \tag{5.38}$$

$$\mathrm{Env'}^{\Lambda k}_{w} = \min_{0 \leq j \leq k} \{\mathrm{Env'}^{\Lambda j}_{\text{right}(w)} - \mathrm{e}^{k-j}_{\text{left}(w)}, \mathrm{Env'}^{j}_{\text{right}(w)} - \mathrm{e}^{\Lambda(k-j)}_{\text{left}(w)}\} \cup \{\mathrm{Env'}^{\Lambda k}_{\text{left}(w)}, \mathrm{Env'}^{\Lambda(k-1)}_{w}\} \tag{5.39}$$

In the case that the number of delayed tasks is greater than the number of available nodes in the subtree of the right side, it is sufficient to retrieve $\mathrm{Env'}^{\Lambda(k-1)}_{w}$ in (5.39), i.e. the lambda energy envelope when at most $k-1$ tasks are delayed.

Finally, the function $\mathrm{Env'}^{\Lambda r}(\Theta^0, \Theta^1, \Lambda^0, \Lambda^1)$ for $\Theta^0 = \{i^0 \mid i \in \Theta\}$ and $\Theta^1 = \{i^1 \mid i \in \Theta\}$ can be computed by computing the value $\mathrm{Env'}^{\Lambda k}_{w}$ at the root node of $(\Theta - \Lambda)^r_U-$tree.

Robust Edge-Finding algorithm for filtering the latest completion times

Analogous to the case for filtering the earliest starting times, the implementation of Edge-Finding for filtering the latest completion times proceeds in two phases.

Detection phase

Algorithm 15 adapts the detection phase of the Edge-Finding for the latest completion times. The algorithm starts with a full $(\Theta - \Lambda)^r_U-$tree, in which all the regular as well as the delayed tasks are scheduled in Θ and $\Lambda = \Lambda^0 \cup \Lambda^1$ is empty. That is,

$$\Theta = \mathcal{I}^0 \cup \mathcal{I}^1, \Lambda^0 = \Lambda^1 = \emptyset$$

The algorithm iterates over the set of all earliest starting times in non-decreasing order. If the filtering of earliest starting times and latest completion times of tasks are both implemented respectively, it is not necessary to test the Overload Checking in algorithm 15. For every $j \in \mathcal{I}$ in non-decreasing order

$$\mathrm{Env'}^{\Lambda r}(\Theta^0, \Theta^1, \Lambda^0, \Lambda^1) < \mathcal{C} \cdot \mathrm{est}_j \tag{5.40}$$

is checked which captures the precedence. Thanks to the structure of the $(\Theta-\Lambda)^r_U-$tree, $\mathrm{Env}'^{\Lambda r}(\Theta^0, \Theta^1, \Lambda^0, \Lambda^1)$ is retrieved from the root of $(\Theta - \Lambda)^r_U-$tree for each $j \in \mathcal{I}$. Line 9 retrieves the task subject to a precedence. It can be implemented in $O(\log(n))$ time by traversing down the tree. The algorithm proceeds by traversing down the $(\Theta - \Lambda)^r_U-$tree from the root. At each inner node w in such a traversal, the algorithm determines which one of the cases in (5.39) satisfy for $\mathrm{Env}'^{\Lambda k}_w$. So long as this task is taken from an $\mathrm{Env}'^{\Lambda k}_f$, where f is a child of w, the algorithm continues down. As soon as the task is taken from an $\mathrm{e}^{\Lambda k}_f$, the algorithm switches to check the cases of (5.21). Similarly, the responsible task subject to a precedence is only one task $i^b \in \Lambda^b$ that is located on a leaf of the $(\Theta - \Lambda)^r_U-$tree and causes (5.36) to be minimized, hence making a precedence. We encode the precedence that i creates in a two dimensional matrix by 0 and 1 columns, named prec, which indicates whether the task is regular or delayed (line 10). For $b \in \{0, 1\}$, $prec[i, b] = \mathrm{est}_j$ for some $j \in \mathcal{I}$ means that the subsets of tasks from $\Theta^0 \cup \Theta^1$ with at most $r - b$ delayed tasks which start no earlier than j are preceded by i^b and i^b cause (5.36) to be minimized. Once the precedence is recorded at line 10, the algorithm unschedules i^b from Λ^b. After the execution of the loop, the tasks corresponding to j is unscheduled from Θ and rather scheduled in Λ^0 and Λ^1.

Algorithm 15: DetectionPhaseOfEdge-FindingForUpperBounds($\mathcal{I}$)

```
1  for i ∈ {1, ..., n} do
2  |  prec[i,0] ← ∞
3  |  prec[i,1] ← ∞
4  Θ ← I
5  Λ^0 ← ∅
6  Λ^1 ← ∅
7  for j ∈ I in non-decreasing order of est_j do
8  |  while Env'^{λr}_{root}(Θ, Λ) < C · est_j do
9  |  |  i^b ← The task in Λ^0 ∪ Λ^1 that minimizes Env'^{Λr}(Θ, Λ)
10 |  |  prec[i, b] ← est_j
11 |  |  Λ^b ← Λ^b \ {i^b}
12 |  Θ ← Θ \ {j}
13 |  Λ^0 ← Λ^0 ∪ {j^0}
14 |  Λ^1 ← Λ^1 ∪ {j^1}
```

Figure 5.4 depicts a trace of the algorithm for the example 5.3.2 in few steps. Figure 5.4a illustrates the initialization step of the algorithm, where the $(\Theta - \Lambda)^r_U-$tree is full. That is, all the regular as well as delayed tasks are scheduled and no task is a candidate yet to create a precedence. In the first iteration the task A is processed and it gets unscheduled. Thus, the two leaves of the $(\Theta - \Lambda)^r_U-$tree corresponding to A^0 and A^1 are modified. The updated status of the node corresponding to A is depicted in figure 5.4b. In the second iteration, B is processed. At this point, $30 < \mathcal{C} \cdot \mathrm{est}_B$, which causes the loop dedicated to detect an existing precedence to execute. Since A^1 is detected as the

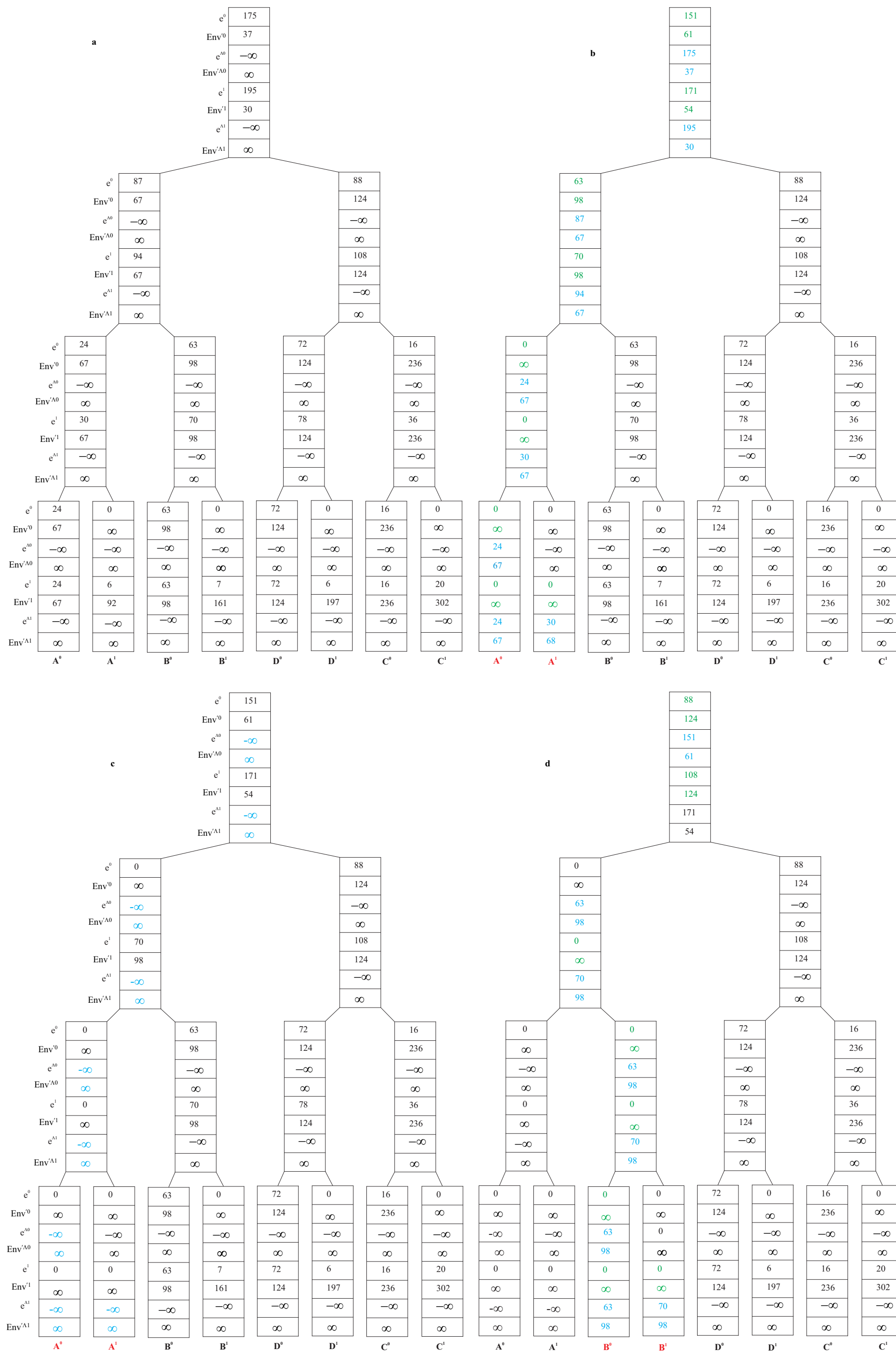


Figure 5.4 – In the figure a, the algorithm starts with a full $(\Theta - \Lambda)^r_U$−tree. In the figure b, A gets unscheduled from Θ and scheduled in Λ. In the figure c, A^1 and then A^0 are detected as responsible tasks and get unscheduled. In the figure d, B is unscheduled.

Task \ b	0	1
A	6	6
B	15	15
C	∞	∞
D	22	22

Table 5.4 – The *prec* array which is obtained after the execution of the algorithm 15 on the instance of example 5.3.2

responsible task, it gets unscheduled. The loop executes again and A^0 is detected as responsible task. Therefore, it gets unscheduled, as illustrated in the figure 5.4c. In the next execution, the loop gets pasted and now B which corresponds to the time point t is unscheduled. Figure 5.4d represents this step. Table 5.4 indicates the prec array as the result of the algorithm 15.

Adjustment phase

For $j \in \mathcal{I}$, we define the *right cut* of j with respect to its est_j to be

$$\mathrm{Rcut}(j) = \{l \in \mathcal{I} : \mathrm{est}_j \leq \mathrm{est}_l\}$$

i.e. $\mathrm{Rcut}(j)$ includes all of the tasks which start no earlier than j.

The adjustment of the latest completion times is done by iterating over the detected precedences which are recorded in the prec array. Let i^b be the task for which a precedence was detected with (5.33) or (5.34) and $\mathrm{prec}[i, b] = \mathrm{est}_j$ for some $j \in \mathcal{I}$. The following formula adjusts the latest completion time of i.

$$\mathrm{lct}_i \leftarrow \min(\mathrm{lct}_i, \min_{\substack{\Omega^0 : \mathcal{I}(\Omega^0) \subseteq \mathrm{Rcut}(j) \setminus \{i\} \\ \Omega^1 : \mathcal{I}(\Omega^1) \subseteq \mathrm{Rcut}(j) \setminus \{i\} \\ \emptyset \neq \Omega^0 \cup \Omega^1 \\ |\Omega^1| \leq r - b \\ \mathrm{rest}(\Omega^0, \Omega^1, c_i) > 0}} \{\max(\mathrm{lct}_{\mathcal{I}(\Omega^0)}, \mathrm{lct}^d_{\mathcal{I}(\Omega^1)}) - \left\lceil \frac{\mathrm{rest}(\Omega^0, \Omega^1, c_i)}{c_i} \right\rceil \}) \tag{5.41}$$

Note that in (5.41), Ω^0 includes the regular tasks and Ω^1 includes the delayed tasks as described in section 5.3.2.

From the condition $\mathrm{rest}(\Omega^0, \Omega^1, c_i) > 0$ we obtain

$$(\mathcal{C} - c_i)(\mathrm{est}_{\Omega^0 \cup \Omega^1}) > (\mathcal{C} - c_i)(\max(\mathrm{lct}_{\mathcal{I}(\Omega^0)}, \mathrm{lct}^d_{\mathcal{I}(\Omega^1)})) - \mathrm{e}_{\Omega^0 \cup \Omega^1} \tag{5.42}$$

The right side of (5.42) describes an equivalent of $\mathrm{Env}'^r(\Theta^0, \Theta^1)$, defined by (5.35), on a resource with capacity $\mathcal{C} - c_i$. This motivates the idea of defining a variant of the latest energy envelope, defined as (3.3), with respect to the capacity c of a task as

$$\mathrm{Env}'^c_\Theta = \min_{\Omega \subseteq \Theta}((\mathcal{C} - c)\,\mathrm{lct}_\Omega - \mathrm{e}_\Omega) \tag{5.43}$$

and a variant of $\text{Env}'^{r}(\Theta^0, \Theta^1)$, defined in (5.35), as

$$\text{Env}'^{cr}(\Theta^0, \Theta^1) = \min_{\substack{\Omega^0 \subseteq \Theta^0 \\ \Omega^1 \subseteq \Theta^1 \\ |\Omega^1| \leq r \\ \mathcal{I}(\Omega^0) \cap \mathcal{I}(\Omega^1) = \emptyset}} ((\mathcal{C} - c) \max(\text{lct}_{\mathcal{I}(\Omega^0)}, \text{lct}^d_{\mathcal{I}(\Omega^1)}) - \text{e}_{\Omega^0} - \text{e}_{\Omega^1}) \tag{5.44}$$

The algorithm 18 indicates our adaption of the adjustment phase of the Edge-Finding for this purpose. For each $c \in F$, we initialize an empty Θ^{cr}_U-tree. This tree is different from Θ^{cr}_L-tree and the structure of its leaves is similar to $(\Theta - \Lambda)^r_U-$tree. That is, there are $2n$ leaves in the tree, associated to the regular and delayed tasks. Each leaf of the Θ^{cr}_U-tree holds the parameters for the energy and latest energy envelope, as well as (5.43) which is computed for $\Theta = \{i\}$. For an inner node w of the Θ^c-tree, (5.43) is computed for the set of all tasks which are included in the subtree rooted at w. This computation is done recursively as below.

$$\text{Env}'^{ck}_w = \min_{0 \leq j \leq k}(\text{Env}'^{cj}_{\text{right}(w)} - \text{e}^{k-j}_{\text{left}(w)}, \text{Env}'^{ck}_{\text{left}(w)}, \text{Env}'^{c(k-1)}_w) \tag{5.45}$$

Θ^{cr}_U-tree develops by processing all prec$[i, b]$ in non-increasing order. The adjustment should be done only if $c_i = c$. If so, at iterating each prec$[i, b] = \text{est}_j$ for some $j \in \mathcal{I}$, all the tasks which cannot start before j are scheduled in the Θ^{cr}_U-tree by adding their regular version to Θ^0 and their delayed version to Θ^1.

Let $\Omega^0 \cup \Omega^1$ be the candidate subset which can give the strongest adjustment in (5.41). Once all the tasks which are eligible to be in $\Omega^0 \cup \Omega^1$ are scheduled in the tree, it is time to identify Ω^0 and Ω^1 and compute the value of the second component of the right side of (5.41). We proceed to compute Ω^0 and Ω^1 in two steps.

The first step is to compute the minimum of $\max(\text{lct}_{\mathcal{I}(\Omega^0)}, \text{lct}^d_{\mathcal{I}(\Omega^1)})$ (or *minlct*) for all Ω^0 and Ω^1 that satisfy (5.41). In the algorithms 16, we consider this idea by taking advantage of the new variant of the latest energy envelope to locate such a node.

The second step is to compute the envelope of the tasks which is done in the algorithm 17 . This is similarly done by dividing the tasks in two groups $\alpha(j, c)$ and $\beta(j, c)$, for the tasks that finish at or after minlct and the tasks which finish before minlct. From the sets $\alpha(j, c)$ and $\beta(j, c)$, the latest energy envelop in the Θ^{cr}_U-tree is computed with

$$\text{Env}(j, c) = \text{Env}_{\alpha(j,c)} - \text{e}_\beta \tag{5.46}$$

where e_β and Env_α are computed in a bottom-up manner, starting from the located task responsible for minlct, and by taking into account that at most r tasks are delayed. Once the envelope is computed, the value which makes the strongest update is adjusted by taking the minimum between the envelope computed in the current iteration and the preceding iterations (line 12 of the algorithm 18). Ultimately, if the task can not start before est_j, it gets scheduled itself.

Algorithm 16: Minlct(tree,bound,c,$\mathcal{C}$,k,b)

```
1  v ← root
2  e[0...k] ← 0⃗
3  minEnv^c ← (C − c).bound
4  k ← k − b
5  while v is not a leaf do
6      branchLeft ← false
7      for j = 0, ..., k do
8          if Env'^{c,j}_{left(v)} − e[k − j] < minEnv^c then
9              v ← left(v)
10             branchLeft ← true
11             break
12     if not branchLeft then
13         v ← right(v)
14         for j = 0, ..., k do
15             e'[j] ← max_{0≤m≤j} e[j] + e^{i−j}_{left(v)}
16         e ← e'
17 return v
```

Finally, it must be mentioned that the same complexity for filtering latest completion times is maintained and the argument is similar.

Algorithm 17: EnvelopeForUpperBound(v,tree,bound,c,k)

```
1  e_α ← [e_v^0, e_v^1, 0, ..., 0] // e_α includes k - 1 entries 0
2  Env'_α ← [Env_v^0, Env_v^1, ∞, ..., ∞] // Env_α includes k - 1 entries ∞
3  e_β ← 0⃗
4  while v is not the root do
5      if v is a right child then
6          e'_β ← 0⃗
7          for j = 0, ..., k do
8              e'_β[j] ← max_{0≤i≤j}(e_β[i] + e^{j−i}_{sibling(v)}, e'_β[j − 1])
9          e_β ← e'_β
10     else
11         Env''_α ← 0⃗
12         e'_α ← 0⃗
13         for j = 0, ..., k do
14             Env''_α[j] ← min(min_{0≤i≤j}(Env'^i_{sibling(v)} − e_α[j − i]), Env'_α[j], Env''_α[j − 1])
15             e'_α[j] ← max(max_{0≤i≤j}(e^i_{sibling(v)} + e_α[j − i]), e'_α[j − 1])
16     Env'_α ← Env''_α
17     e_α ← e'_α
18     v ← parent(v)
19 return min_{0≤i≤k}(Env'_α[k − i] − e_β[i])
```

Algorithm 18: AdjustmentOfUpperBounds(prec,r,$\mathcal{C}$)

```
1  F ← The capacities of the tasks for which a precedence was detected
2  for c ∈ F do
3      Θ ← ∅
4      upd ← [∞, ∞]
5      t' ← ∞
6      for i^b ∈ {j^b | prec[j, b] < ∞ ∧ c_i = c} in non-increasing order of prec[i, b] do
7          t ← prec[i, b] for some j ∈ 𝓘
8          Θ ← Θ ∪ (Rcut(j) \ Rcut(t')) \ {i}
9          v ← MinIct(Θ, t, c, 𝒞, r, b)
10         Env' ← EnvelopeForUpperBound(v, Θ, r − b)
11         diff ← ⌊(Env' − (𝒞 − c)t/c⌋
12         upd[b] ← min(upd[b],diff)
13         lct_i ← min(lct_i,upd[b])
14         if est_i ≥ t then
15             Θ ← Θ ∪ {i}
16         t' ← t
```

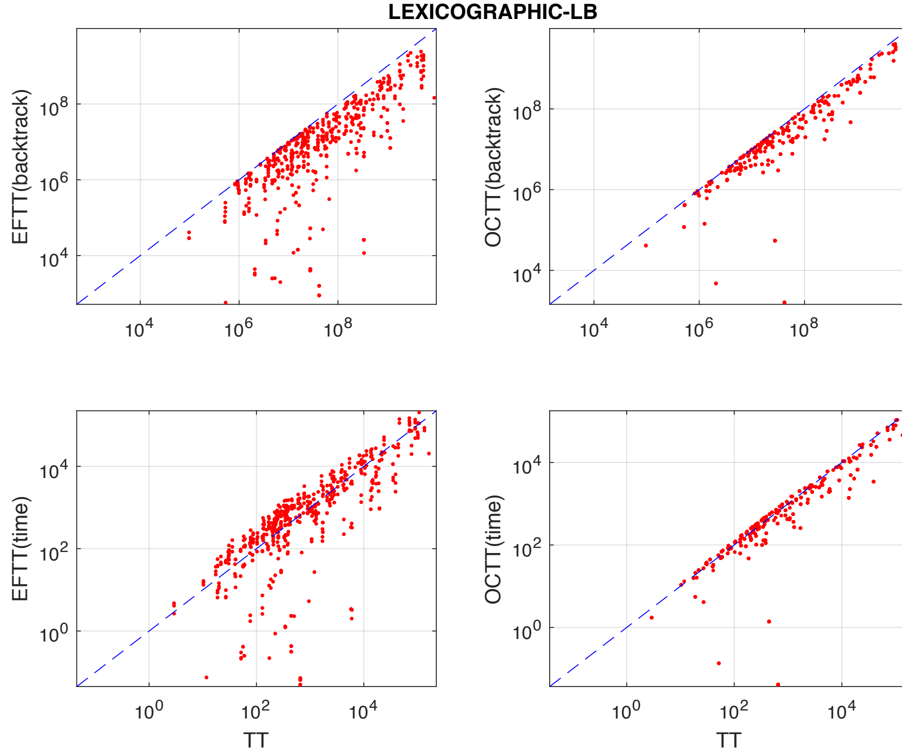


Figure 5.5 – The logarithmic scale graphs as the results of running Time-Tabling and Overload Checking (denoted OC-TT) or Time-Tabling and Edge-Finding(denoted EF-TT) with lexicographic heuristics in terms of backtrack numbers as well as elapsed times. The horizontal axis corresponds to the Time-Tabling and the vertical axis corresponds to Overload Checking or Edge-Finding.

5.4 Experiments

The experiments were carried out on a 2.0 GHz Intel Core i5, with Choco version 3.3.1. We tested our algorithms against the BL suite of the RCPSP instances [10]. This benchmark consists of 40 highly cumulative instances with either 20 or 25 tasks, subject to precedence constraints, to be executed on several resources. We minimize the makespan. For the delay attributes associated to every task i, we uniformly generated random numbers in $[0, 2 \cdot p_i]$. We used three different heuristics: Lexicographic, DomOverWDeg [14], and Impact Based Search [65]. For these three heuristics, the figures 5.5, 5.6 and 5.7 respectively illustrate a logarithmic scale representation of the number of backtracks and the elapsed time measurements when a combination of Time-Tabling and Overload Checking or Time-Tabling and Edge-Finding are implemented. For the Time-Tabling algorithm, we used the same implementation as in Derrien et al. [21] that we obtained from the authors.

As the graphs verify, our algorithms lead to fewer backtracks. The results are much more significant for the Edge-Finding. This is due to the fact that the Edge-Finding filters the domains while the Overload Checking only triggers backtracks. It appears that the heuristic chosen for solving the problem also affects the results. Thereby, we selected the lexicographic heuristic as one of our heuristics

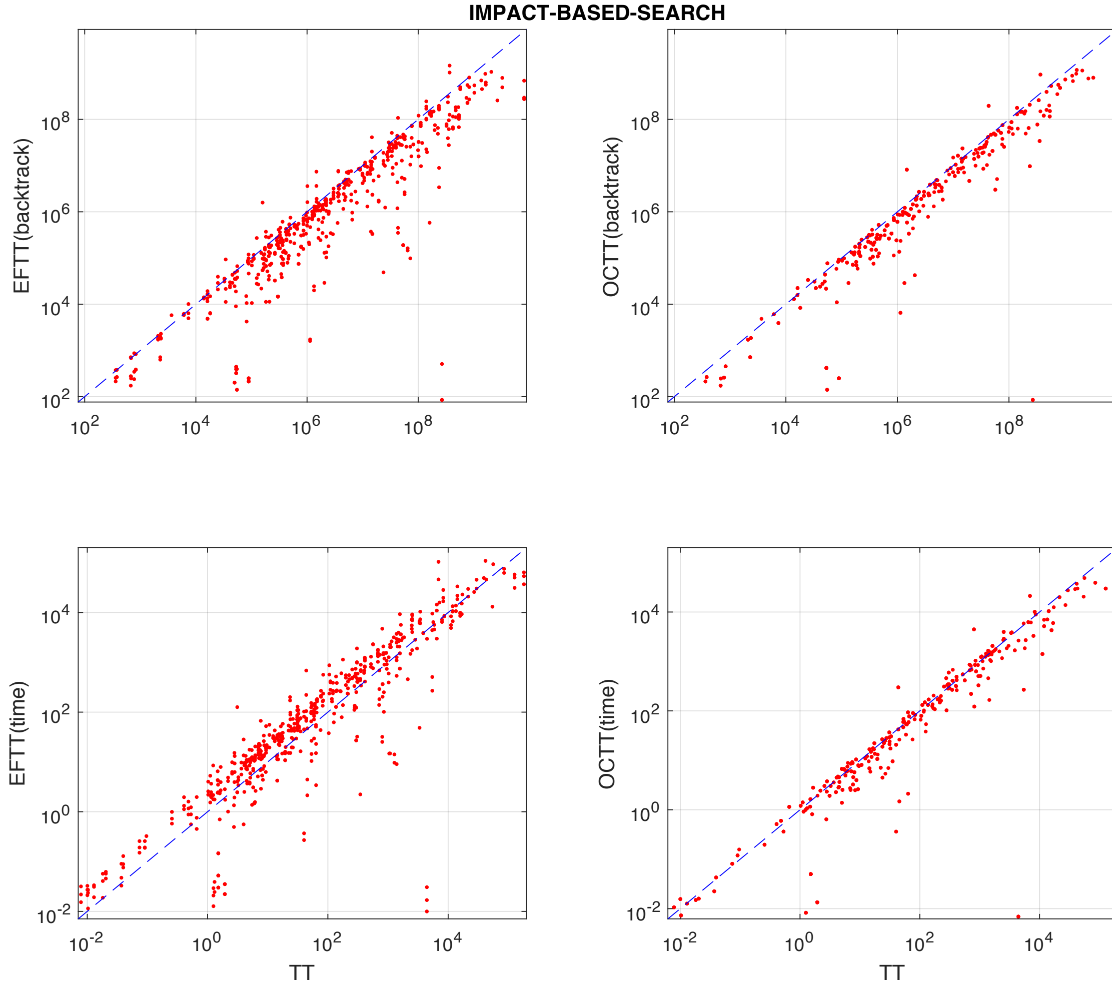


Figure 5.6 – The logarithmic scale graphs as the results of running Time-Tabling and Overload Checking (denoted OC-TT) or Time-Tabling and Edge-Finding(denoted EF-TT) with impact based search heuristics in terms of backtrack numbers as well as elapsed times. The horizontal axis corresponds to the Time-Tabling and the vertical axis corresponds to Overload Checking or Edge-Finding.

so that the heuristic chosen does not impinge our objective, which is proving that the combination of our algorithms with the Time-Tabling provides a stronger filtering. The lexicographic heuristic is certainly not the best heuristic to solve scheduling problems, but it allows seeing how much pruning a new filtering algorithm achieves. The combination of our algorithms with Time-Tabling improves the resolution times for many instances. This fact can differ from one heuristic to another. Overall, among the state of the art heuristics, IMPACT-BASED-SEARCH performs more efficiently, as it leads to fewer backtracks and faster computation times for many instances.

We also compared the implementation of our Edge-Finding algorithm with a conjunction of n CUMULATIVE constraints, as described in section 2.2. The performances for the conjunction of n CUMULATIVE constraints are so much worse in terms of time that we omit to report them.

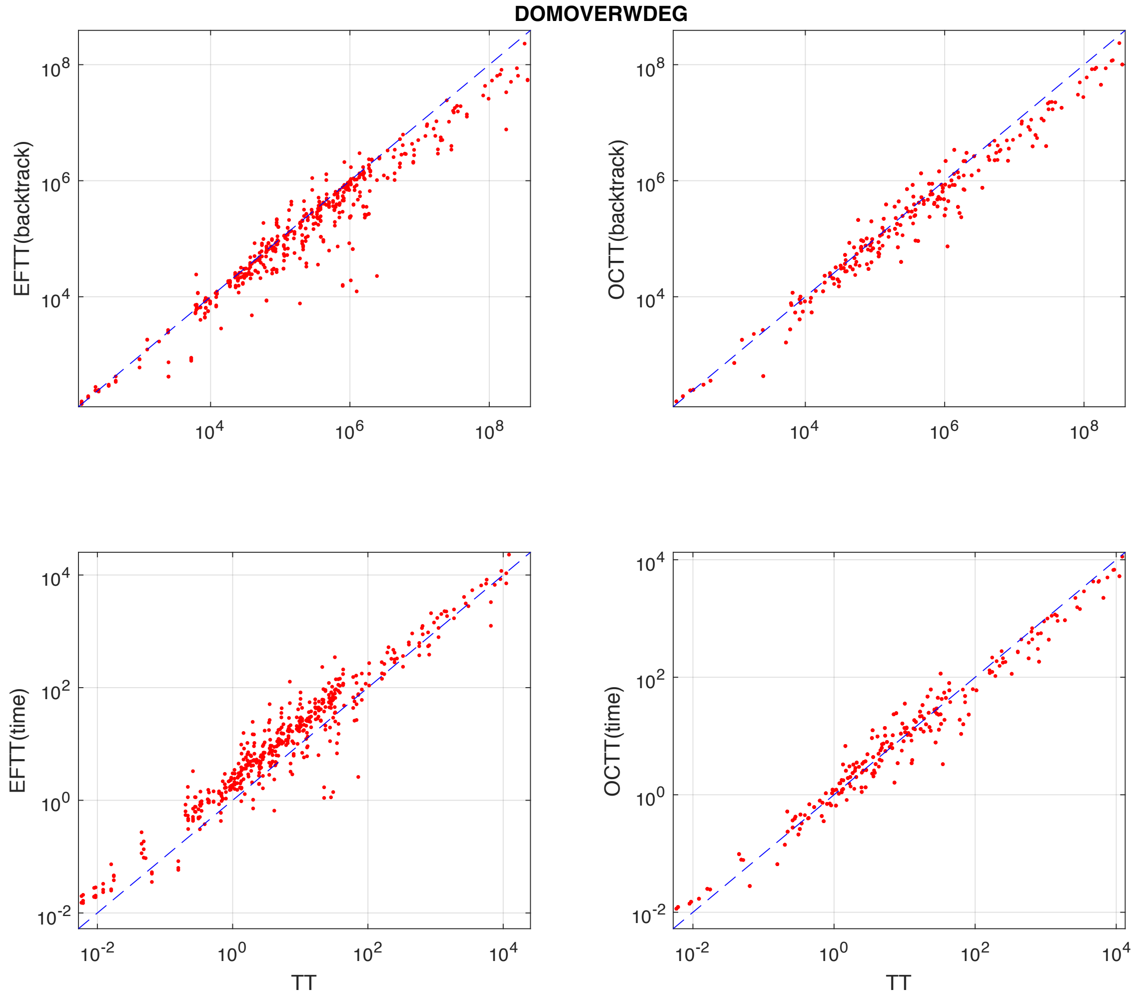


Figure 5.7 – The logarithmic scale graphs as the results of running Time-Tabling and Overload Checking (denoted OC-TT) or Time-Tabling and Edge-Finding(denoted EF-TT) with domoverwdeg heuristics in terms of backtrack numbers as well as elapsed times. The horizontal axis corresponds to the Time-Tabling and the vertical axis corresponds to Overload Checking or Edge-Finding.

5.5 Conclusion

We adapted the state of the art algorithms for Overload Checking and Edge-Finding in robust cumulative scheduling problems. The experimental results demonstrate a stronger filtering when our algorithms are combined with Time-Tabling.

Chapter 6

Variants of Multi-Resource Scheduling Problems with Equal Processing Times

In the preceding chapters we outlined the process of reducing the search space by devising efficient or novel filtering algorithms for the scheduling problems which are NP-hard. However, as it is pointed out in section 2.3, there exist scheduling problems that can be encountered by the industry for which rarely scheduling algorithms do exist. For instance, the number of resources can fluctuate over time (e.g. there could be more employees working by day than by night), the cost of a resource can also fluctuate over time (the electricity can be cheaper by night than by day). Such variations require us to modify existing algorithms for conventional scheduling. In this section we rather tackle $P \mid \mathrm{est}_j; p_j = p; \mathrm{lct}_j \mid \gamma$, mentioned also in (2.1), *i.e,* the problem of non-preemptive scheduling of a set of tasks of the constant duration p over m machines with given release and deadline times.

Recalling that $o_i = \mathrm{lst}_i + 1$, a solution to this problem is an assignment of the starting times S_i which satisfies the following constraints

$$\mathrm{est}_i \leq S_i < o_i \qquad \forall\, i \in \mathcal{I} \tag{6.1}$$

$$|\{i : t \leq S_i < t + p\}| \leq m \qquad \forall\, t \in [\mathrm{est}_{\mathcal{I}}, \mathrm{lct}_{\mathcal{I}} - p] \tag{6.2}$$

The problem is sometimes reformulated by dividing all time points by p, resulting in tasks with unit processing times [74, 75]. However, this formulation does not make the problem easier to solve, as release times and deadlines lose their integrality. Without this integrality, greedy algorithms commonly used to solve the problem when $p = 1$, become incorrect. Indeed, when the greedy scheduling algorithms choose to start a task i, they assume that no other tasks arrive until i is completed. This assumption does not hold if the release times can take any rational value.

We explore several variations of the particular scheduling problem, denoted by the three field notation (2.1). Firstly, we solve the problem when the number of machines fluctuates over time. This models situations where there are fewer operating machines during night shifts or when fewer employees

can execute tasks during vacation time or holidays. Then, we consider the problem with different objective functions. For an arbitrary function $w_i(t)$ associated to task i that maps a time point to a cost, we prove that minimizing $\sum_{i=1}^{n} w_i(S_i)$ is NP-Hard. This function is actually very general and can encode multiple well known objective functions. We study the case where all tasks share the same function $w(t)$. This models the situtation where the cost of using the resource fluctuates with time. This is the case, for instance, with the price of electricty. Executing any task during peak hours is more expensive than executing the same task during a period when the demand is low. We show that minimizing $\sum_{i=1}^{n} w(t)$ can be done in pseudo-polynomial time and propose improvements when $w(t)$ is monotonic or periodic. The periodicty of the cost function is a realistic assumption as high and low demand periods for electricity have a predictable periodic behavior. Finally, we point out how the problem is solved in polynomial time with the objective of minimizing maximum lateness.

This chapter is organized as follows. Section 6.1 solves the case where the number of machines fluctuates at specific times and shows how to adapt an existing algorithm for this case, while preserving polynomiality. Section 6.2 shows that minimizing $\sum_{i=1}^{n} w_i(S_i)$ is NP-Hard. Sections 6.3 and 6.4 consider a unique cost function $w(t)$ that is either monotonic or periodic and present polynomial time algorithms for these cases. Finally, we show how to adapt a polynomial time algorithms for minimizing maximum lateness.

6.1 Variety of machine numbers through the time

Consider (2.1) with the assumption that the number of machines fluctuates over time. Let $T = [(t_0, m_0), \ldots, (t_{|T|-1}, m_{|T|-1})]$ be a sequence where t_i's are the time points at which the fluctuations occur and they are sorted in chronological order and m_i machines are available within the time interval $[t_i, t_{i+1})$. This time interval is the union of a (possibly empty) interval and an open-interval: $[t_i, t_{i+1}) = [t_i, t_{i+1} - p] \cup (t_{i+1} - p, t_{i+1})$. A task starting in $[t_i, t_{i+1} - p]$ is guaranteed to have access to m_i machines throughout its execution, whereas a task starting in $(t_{i+1} - p, t_{i+1})$ encounters the fluctuation of the number of machines before completion. Therefore, the number of tasks which can start in the interval $(t_{i+1} - p, t_{i+1})$ is no more than the minimum number of machines available within the interval in which they execute. In general, a task can encounter multiple fluctuations of the number of machines througout its execution. Let $\alpha(t) = \max\{t_j \in T \mid t_j \leq t\}$ be the last time the number of machines fluctuates before time t. At most $M(t)$ tasks can start at time t.

$$M(t) = \min\{m_i \mid t_i \in [\alpha(t), t + p)\}. \tag{6.3}$$

From (6.3), we conclude that no more than $\max_{t' \in [t, t+p)} M(t')$ tasks can start in the interval $[t, t+p)$. Recall that the constraint (2.8) enforces the number of tasks starting in a time lag of size p not to be greater than m resources. According to (6.3), one can rewrite the constraint (2.8) as

$$x_{t+p} - x_t \leq \max_{t \leq t' < t+p} M(t') \tag{6.4}$$

and update the weight function (2.11) of the scheduling graph as

$$w'(a,b) = \begin{cases} \max_{a \le t' < a+p} M(t') & \text{if } a+p=b \\ n & \text{if } a = \text{est}_{\mathcal{I}} \wedge b = o_{\mathcal{I}} \\ -\left|\{k : b \le \text{est}_k \wedge o_k \le a\}\right| & \text{if } a \ge b \end{cases} \quad (6.5)$$

It remains to show how the algorithm presented in [48] can be adapted to take into account the fluctuating number of machines. This algorithm is based on the Bellman-Ford algorithm and maintains a vector $d^{-1}[0..n]$ such that $d^{-1}[i]$ is the latest time point reachable at distance at most $-i$ from the node $o_{\mathcal{I}}$. In other words, all nodes whose label is a time point in the semi-open interval $(d^{-1}[i+1], d^{-1}[i]]$ are reachable at distance $-i$ from node $o_{\mathcal{I}}$. Let a be a node in $(d^{-1}[i+1], d^{-1}[i]]$ and consider the edge $(a, a+p)$ of weight $w'(a, a+p)$. Upon processing this edge, the algorithm updates the vector by setting $d^{-1}[i - w'(a,b)] \leftarrow \max(d^{-1}[i - w'(a,b)], b)$, i.e. the rightmost node accessible at distance $-i + w'(a,b)$ is either the one already found, or the node $a+p$ that is reachable through the path to a of distancre $-i$ followed by the edge $(a, a+p)$ of distance $w'(a, a+p)$.

To efficiently proceed to this update, the algorithm evaluates the function $w'(a, a+p)$ in two steps. The first step transforms T to a sequence $T' = [(t'_0, m'_0), (t'_1, m'_1), \ldots]$ such that $M(t) = m'_i$ for every $t \in [t'_i, t'_{i+1})$. The second step transforms the sequence T' into a sequence $T'' = [(t''_0, m''_0), (t''_1, m''_1), \ldots]$ such that $w'(t, t+p) = m''_i$ for all $t \in [t''_i, t''_{i+1})$. Interestingly, both steps execute the same algorithm.

To build the sequence T', one needs to iterate over the sequence T and find out, for every time window $[t, t+p)$, the minimum number of available machines inside that time window. If a sequence of consecutive windows such as $[t, t+p), [t+1, t+p+1), [t+2, t+p+2), \ldots$ have the same minimum number of available machines, then only the result of the first window is reported. This is a variation of the *minimum on a sliding window problem* [34] where an instance is given by an array of numbers $A[1..n]$ and a window length p. The output is a vector $B[1..n-p+1]$ such that $B_i = \min\{A_i, A_{i+1}, \ldots, A_{i+p-1}\}$. The algorithm that solves the minimum on a sliding window problem can be slightly adapted. Rather than taking as input the vector A that contains, in our case, many repetitions of values, it can simply take as input a list of pairs like the vector T and T' which indicate the value in the vector and until which index this value is repeated. The same compression technique applies for the output vector. This adaptation can be done while preserving the linear running time complexity of the algorithm.

Once computed, the sequence T' can be used as input to the *maximum on a sliding window problem* to produce the final sequence T''. Finally, the algorithm 19 simultaneously iterates over the sequence T'' and the vector d^{-1} to relax the edges in $O(|T|+n)$ time. Since relaxing forward edges occurs at most $O(\min(1, \frac{p}{m})n)$ times [48], the overall complexity to schedule the tasks is $O(\min(1, \frac{p}{m})(|T|+n)n)$.

Figure 6.1 is an illustrative example of a trace of the algorithm 19 for a set of tasks with equal processing times $p = 2$. In this example, $T = [(1,3), (3,5), (4,2), (6,6)]$, as illustrated in figure 6.1a. The figures 6.1b and 6.1c respectively describes how the sequences T' and T'' are computed.

Algorithm 19: RelaxForwardEdges($[(t''_1, m''_1), \ldots, (t''_{|T''|}, m''_{|T''|})], d^{-1}[0..n], p$)

1. $t \leftarrow \text{est}_{\mathcal{I}}$, $i \leftarrow n$, $j \leftarrow 0$
2. **while** $i > 0 \lor j < |T''|$ **do**
3.   **if** $i - m''_j > 0$ **then** $d^{-1}[i - m''_j] \leftarrow \max(d^{-1}[i - m''_j], t + p)$
4.   **if** $j = |T''| \lor (i > 0 \land d^{-1}[i-1] < m''_{j+1})$ **then**
6. $i \leftarrow i - 1$
7. $t \leftarrow d^{-1}[i]$
8.   **else**
9. $j \leftarrow j + 1$
10. $t \leftarrow t''_j$

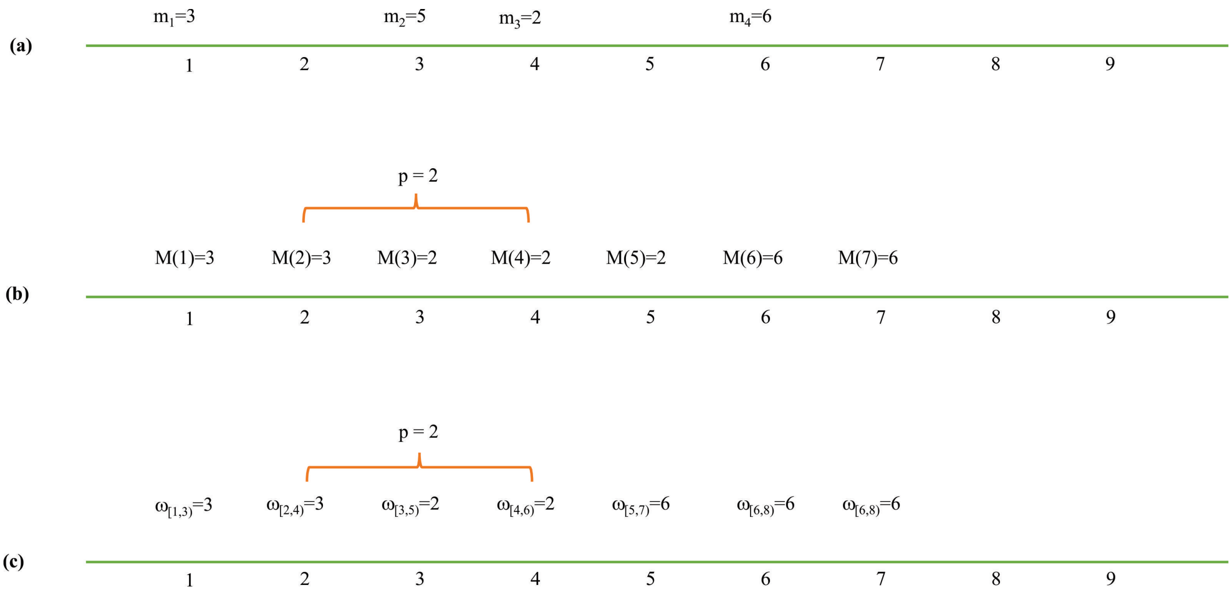


Figure 6.1 – A trace of the algorithm 19, where $p = 2$ and $T = [(1,3),(3,5),(4,2),(6,6)]$. The sequences T' and T'' are constructed, as represented at lines (b) and (c).

6.2 General Objective Function

As mentioned in chapter 2, an alternative common objective function is to minimize the sum of the completion times. Recall from chapter 2 that when the tasks have equal processing times, the objectives of minimizing the sum of completion times and minimizing the sum of starting times are equivalent. In the following we consider minimizing costs per task and per time, i.e. $\sum_{i,t} w_i(S_i)$ for arbitrary functions $w_i(t)$. Such an objective function depends on the starting time of the task. In the industry, this objective function can be used to model a cost that increases as the execution of a task is delayed, such as $w_i(t) = t - \text{est}_i$.

Lemma 6. Minimizing $\sum_{i,t} w_i(S_i)$ for arbitrary functions $w_i(t)$ is NP-Hard.

Proof. We proceed with a reduction from the INTER-DISTANCE constraint. Recall from section 2.2.3

that this constraint states that the predicate INTER-DISTANCE($[X_1, \ldots, X_n], p$) is true if and only if $|X_i - X_j| \geq p$ holds whenever $i \neq j$. Let $S_1, \ldots, S_n$ be n sets of integers. Deciding whether there exists an assignment for the variables $X_1, \ldots, X_n$ such that $X_i \in S_i$ and INTER-DISTANCE($[X_1, \ldots, X_n], p$) hold is NP-Complete [3]. We create one task per variable X_i with release time $\text{est}_i = \min(S_i)$, latest starting time $\text{lst}_i = \max(S_i)$, processing time p, and a cost function $w_i(t)$ equal to 0 if $t \in S_i$ and 1 otherwise. There exists a schedule with objective value $\sum_{i,t} w_i(S_i) = 0$ iff there exists an assignment with $X_i \in S_i$ that satisfies the predicate INTER-DISTANCE, hence minimizing $\sum_{i,t} w_i(S_i)$ is NP-Hard. □

The NP-hardness of this problem motivates the idea of studying specializations of this objective function in order to seek if polynomial time algorithms can be derived.

6.3 Scheduling problems with monotonic objective functions

Lemma 6 indicates that the general objective of minimizing costs per task and per time is NP-Hard. In the following we show if the cost is only dependent on the time when the tasks execute, one can derive polynomial time algorithms for particular objective functions. As a first problem that is a specialization of $P \mid \text{est}_j; p_j = p; \text{lct}_j \mid \gamma$ and leads to a polynomial time algorithm we study the case that the cost function is monotonic.

Let $w(t) : \mathbb{Z} \to \mathbb{Z}$ be an increasing function, i.e. $w(t) + 1 \leq w(t+1)$ for any t. We prove that a schedule that minimizes $\sum_i S_i$ also minimizes $\sum_i w(S_i)$. Theorem 1 shows how to obtain a solution that minimizes $\sum_i S_i$. Lemma 7 shows that this solution also minimizes other objective functions. Recall that h_t is the number of tasks starting at time t.

Lemma 7. The schedule obtained with Theorem 1 minimizes $\sum_{a=t}^{o_{\max}-1} h_a$ for any time t.

Proof. Let $(a_1, a_2), (a_2, a_3), \ldots, (a_{k-1}, a_k)$, with $a_1 = o_{\mathcal{I}}$ and $a_k = t$, be the edges on the shortest path from $o_{\mathcal{I}}$ to t in the scheduling graph. According to the definition of a shortest path, we have $\delta(o_{\mathcal{I}}, a_i) + w'(a_i, a_{i+1}) \geq \delta(o_{\mathcal{I}}, a_{i+1})$. Since $x_t = n + \delta(o_{\mathcal{I}}, t)$, therefore $w'(a_i, a_{i+1}) \geq x_{a_{i+1}} - x_{a_i}$ and we obtain $\delta(o_{\mathcal{I}}, t) = \sum_{i=1}^{k-1} w'(a_i, a_{i+1}) \geq \sum_{i=1}^{k-1} (x_{a_{i+1}} - x_{a_i}) = x_t - x_{o_{\mathcal{I}}}$. This shows that the difference $x_t - x_{o_{\mathcal{I}}}$ is at most $\delta(o_{\mathcal{I}}, t)$ for any schedule. It turns out that by setting $x_t = n + \delta(o_{\mathcal{I}}, t)$, the difference $x_t - x_{o_{\mathcal{I}}} = \delta(o_{\mathcal{I}}, t) - \delta(o_{\mathcal{I}}, o_{\mathcal{I}}) = \delta(o_{\mathcal{I}}, t)$ reaches its maximum and therefore, $x_{o_{\max}} - x_t = \sum_{a=t}^{o_{\max}-1} h_a$ is maximized. □

Theorem 5. The schedule of theorem 1 minimizes $\sum_{i=1}^{n} w(S_i)$ for any increasing function $w(t)$.

Proof. Consider the following functions that differ by their parameter a.

$$w_a(t) = \begin{cases} w(t) & \text{if } t < a \\ w(a) + t - a & \text{otherwise} \end{cases} \tag{6.6}$$

The function $w_a(t)$ is identical to $w(t)$ up to point a and then increases with a slope of one. As a base case of an induction, the schedule described in Theorem 1 minimizes $\sum_{i=1}^{n} S_i$ and therefore minimizes $\sum_{i=1}^{n} w_{\mathrm{est}_{\mathcal{I}}}(S_i)$. Suppose that the algorithm minimizes $\sum_{i=1}^{n} w_a(S_i)$, we prove that it also minimizes $\sum_{i=1}^{n} w_{a+1}(S_i)$. Consider the function

$$\Delta_a(t) = \begin{cases} 0 & \text{if } t \leq a \\ w(a+1) - w(a) - 1 & \text{otherwise} \end{cases} \tag{6.7}$$

and note that $w_a(t) + \Delta_a(t) = w_{a+1}(t)$. For all t, since $w(t+1) - w(t) \geq 1$, we have $\Delta_a(t) \geq 0$.

If $w(a+1) - w(a) = 1$ then $\Delta_a(t) = 0$ for all t and therefore $w_a(t) = w_{a+1}(t)$. Since the algorithm returns a solution that minimizes $\sum_{i=1}^{n} w_a(S_i)$, it also minimizes $\sum_{i=1}^{n} w_{a+1}(S_i)$.

If $w(a+1)-w(a) > 1$, a schedule minimizes the function $\sum_{i=1}^{n} \Delta_a(S_i)$ if and only if it minimizes the number of tasks starting after time a. From Lemma 7, the schedule described in Theorem 1 achieves this. Consequently, the algorithm minimizes $\sum_{i=1}^{n} w_a(S_i)$, it minimizes $\sum_{i=1}^{n} \Delta_a(S_i)$, and therefore, it minimizes $\sum_{i=1}^{n} w_a(S_i) + \sum_{i=1}^{n} \Delta_a(S_i) = \sum_{i=1}^{n} w_{a+1}(S_i)$.

By induction, the algorithm minimizes $\sum_{i=1}^{n} w_{\infty}(S_i) = \sum_{i=1}^{n} w(S_i)$. □

If the cost $w(t)$ function is decreasing, i.e. $w(t) - 1 \geq w(t+1)$, it is possible to minimize $\sum_{i=1}^{n} w(t)$ by solving a transformed instance. For each task i in the original problem, one creates a task i with release time $\mathrm{est}'_i = -\mathrm{lst}_i$ and latest starting time $\mathrm{lst}'_i = -\mathrm{est}_i$. The objective function is set to $w'(t) = -w(t)$ which is an increasing function. From a solution S'_i that minimizes $\sum_{i=1}^{n} w'(S'_i)$, one retrieves the original solution by setting $S_i = -S'_i$.

6.4 Scheduling problems with periodic objective functions

This section aims at providing a pseudo-polynomial algorithm when the cost function $w(t)$ in (2.1) is a periodic function which increases throughout each period. That is, $w(t) < w(t+x)$ for $1 \leq x < W$ and $w(t) = w(t+W)$.

6.4.1 Scheduling problem as a network flow

Theorem 1 of chapter 2 shows that computing the shortest paths in the scheduling graph can minimize the sum of the completion times. We show that computing, in pseudo-polynomial time, a flow in the scheduling graph can minimize $\sum_{i=1}^{n} w(S_i)$ for an arbitrary function $w(t)$.

The objective function (2.6) can be modified to take into account the function w. We therefore minimize $\sum_{t=\mathrm{est}_{\mathcal{I}}}^{o_{\mathcal{I}}-1} w(t)h_t$. After proceeding to the change of variables $x_t = \sum_{i=\mathrm{est}_{\mathcal{I}}}^{t-1} h_i$, we obtain

i	est_i	o_i
1	4	8
2	1	4
3	1	6
4	1	9
5	1	6

$m = 2$
$p = 2$
$w(t) = t \bmod 3$

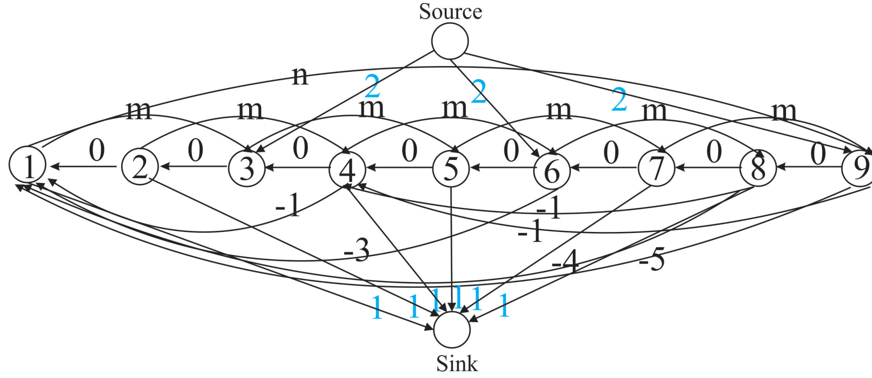


Figure 6.2 – A network flow with 5 tasks. The cost on the forward, backward, and null edges are written in black. These edges have unlimited capacities. The capacities of the nodes from the source and to the sinks are written in blue. These edges have a null cost.

$\sum_{t=\text{est}_\mathcal{I}}^{o_\mathcal{I}-1} w(t)(x_{t+1} - x_t)$ which is equivalent to

$$\text{maximize } w(\text{est}_\mathcal{I})x_{\text{est}_\mathcal{I}} + \sum_{t=\text{est}_\mathcal{I}+1}^{o_\mathcal{I}-1} \left(w(t) - w(t-1)\right) x_t - w(o_\mathcal{I} - 1)x_{o_\mathcal{I}}$$

We use this new objective function with the original constraints of the problem given by equations (2.8) to (2.10). This results in a linear program of the form $\max\{\vec{w}^T x \mid A\vec{x} \leq \vec{b}, \vec{x} \lesseqgtr 0\}$ which has for dual $\min\{\vec{b}^T y \mid A^T \vec{y} = \vec{w}, y \geq 0\}$. Note that every row of matrix A has exactly one occurrence of value 1, one occurrence of the value -1, and all other values are null. Consequently, A^T is a *network matrix* and the dual problem $\min\{\vec{b}^T y \mid A^T \vec{y} = \vec{w}, y \geq 0\}$ is a min-cost flow problem.

Following Section 2.3.3, we reconstruct the graph associated to this network flow which yields the scheduling graph augmented with a source node and a sink node. An edge of capacity $w(\text{est}_\mathcal{I})$ connects the node $\text{est}_\mathcal{I}$ to the sink. An edge of capacity $w(o_\mathcal{I} - 1)$ connects the source node to the node $o_\mathcal{I}$. For the nodes t such that $\text{est}_\mathcal{I} < t < o_\mathcal{I}$, an edge of capacity $w(t-1) - w(t)$ connects the source node to node t whenever $w(t-1) > w(t)$ and an edge of capacity $w(t) - w(t-1)$ connects the node t to the sink node whenever $w(t-1) < w(t)$. All other edges in the graph (forward, backward, and null edges) have an infinite capacity. Figure 6.2 illustrates an example of such a graph.

The computation of a min-cost flow gives rise to a solution for the dual problem. To convert the

solution of the dual to a solution for the primal (i.e. an assignement of the variables x_t), one needs to apply a well known principle in network flow theory [2]. Let $\delta(a,b)$ be the shortest distance from node a to node b in the *residual graph*. The assignement $x_t = \delta(o_{\mathcal{I}}, t)$ is an optimal solution of the primal. The variable x_t is often called *node potential* in network theory.

Consider a network flow of $|V(N)|$ nodes, $|E(N)|$ edges, a maximal capacity of U, and a maximum absolute cost of Q. The successive shortest path algorithm computes a min-cost flow with $O(|V(N)|U)$ computations of a shortest path that each executes in $O(|E(N)|\sqrt{|V(N)|}\log Q)$ time using Goldberg's algorithm [32]. Let $\Delta c = \max_t |w(t) - w(t-1)|$ be the maximum cost function fluctuation and $H = o_{\mathcal{I}} - \text{est}_{\mathcal{I}}$ be the horizon. In the scheduling graph, we have $|V(N)| \in O(H)$, $|E(N)| \in O(H + n^2)$, $Q \in O(n)$, and $U = \Delta c$. Therefore, the overall running time complexity to find a schedule is $O((H - p + n^2)(H)^{3/2}\Delta c \log n)$.

Note that the efficiency of this algorithm is pseudo-polynomial as it is proportional to the size of the horizon H and the maximum cost function fluctuation Δc. Next section discusses an objective function for which an algorithm with a more efficient complexity can be achieved.

6.4.2 Periodic objective function formulated as a network flow

In many occasions, one encounters the problem of minimizing $\sum_{i=1}^{n} w(S_i)$ where $w(S_i)$ is a periodic function, i.e. a function where $w(t) = w(t + W)$ for a period W. Moreover, within a period, the function is increasing. An example of such a function is the function $w(t) = t \bmod 7$. If all time points correspond to a day, the objective function ensures that all tasks are executed at their earliest time in a week. In other words, it is better to wait for Monday to start a task rather than executing this task over the weekend. In such a situation, it is possible to obtain a more efficient time complexity than the algorithm presented in the previous section.

Without loss of generality, we assume that the periods start on times kW for $k \in \mathbb{N}$ which implies that the function $w(t)$ is only decreasing between $w(kW - 1)$ and $w(kW)$ for some $k \in \mathbb{N}$. In the network flow from Section 6.4.1, only the time nodes kW have an incoming edge from the source. We use the algorithm from [48] to compute the shortest distance from every node kW to all other nodes. Thanks to the null edges, distances can only increase in time, i.e. $\delta(kW, t) \leq \delta(kW, t+1)$, and because of the edge $(\text{est}_{\mathcal{I}}, o_{\mathcal{I}})$ of cost n and the inexistence of negative cycles, all distances lie between $-n$ and n. Therefore, the algorithm outputs a list of (possibly empty) time intervals $[a^k_{-n}, b^k_{-n}), [a^k_{-n+1}, b^k_{-n+1}), \ldots, [a^k_n, b^k_n)$ where for any time $t \in [a^k_d, b^k_d)$, $\delta(kW, t) = d$. The min-cost flow necessarily pushes the flow along these shortest paths. We simply need to identify which shortest paths the flow follows.

There are $w(kW-1) - w(kW)$ units of flow that must circulate from node kW and $w(t) - w(t-1)$ units of flows that must arrive to node t, for any t that is not a multiple of W. In order to create a smaller graph with fewer nodes, we aggregate time intervals where time points share common properties. We consider the sorted set S of time points a^k_i and b^k_i. Let t_1 and t_2 be two consecutive time

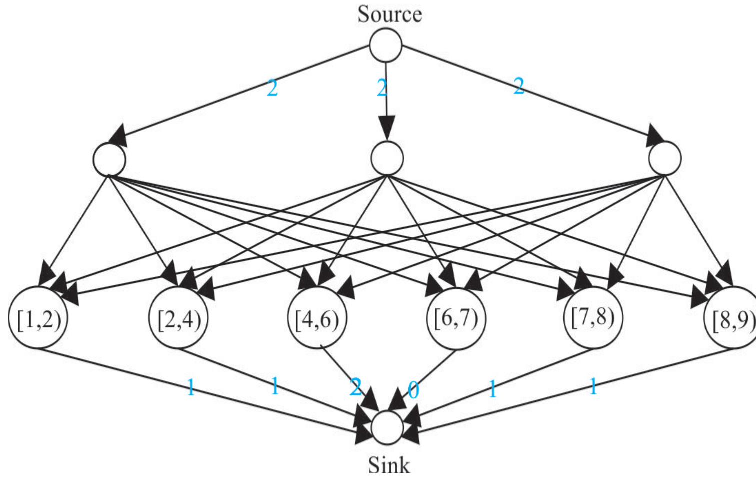


Figure 6.3 – The compressed version of the graph on Figure 6.2. The numbers over the edges connected to the source and sinks stand for the capacities and the costs are omitted.

points in this set. All time points in the interval $[t_1, t_2)$ are at equal distance from the node kW, for any $k \in \mathbb{N}$. The amount of units of flow that must reach the sink from the nodes in $[t_1, t_2)$ is given by

$$\sum_{j=t_1}^{t_2-1} \max(w(j) - w(j-1), 0) = w(t_2 - 1) - w(t_1 - 1) + (\lfloor \frac{t_2 - 1}{W} \rfloor - \lceil \frac{t_1}{W} \rceil + 1)(w(W-1) - w(0)) \tag{6.8}$$

Consequently, we create a graph, called the *compressed graph*, with one source and one sink node. There is one node for each time point kW for $\frac{\mathrm{est}_{\mathcal{I}}}{W} \leq k \leq \frac{o_{\mathcal{I}}}{W}$. There is an edge between the source node and a node kW with capacity $w(kW-1) - w(kW)$. For any two consecutive time points t_1, t_2 in S there is a time interval node $[t_1, t_2)$. An edge whose capacity is given by equation (6.8) connects the interval node $[t_1, t_2)$ to the sink. Finally, a node kW is connected to an interval node $[t_1, t_2)$ with an edge of infinite capacity and a cost of $\delta(kW, t_1)$. Figure 6.3 shows the compressed version of the graph on figure 6.2.

Computing a min-cost flow in this network simulates the flow in the scheduling graph. Indeed, a flow going through an edge $(kW, [t_1, t_2))$ in the compressed graph is equivalent, in the scheduling graph, to a flow leaving the source node, going to the node kW, going along the shortest path from node kW to a time node $t \in [t_1, t_2)$, and reaching the sink.

Theorem 6. To every min-cost flow in the compressed graph corresponds a min-cost flow in the scheduling graph.

Proof. Let G be the scheduling graph and G' be the compressed graph. Let Y' denote a min-cost flow in G'. We show how to obtain a min-cost flow Y in G whose cost is the same as the cost of Y'.

Consider an edge $e_j = (kW, [t_1, t_2))$ in G' which conveys a positive amount of flow, say f. In the scheduling graph G, it is possible to push f units of flow along the shortest paths from kW to the nodes within the interval $[t_1, t_2)$. It suffices to see how one can retrieve Y from Y', presuming it is initially null. This is done by considering all incoming flows to $[t_1, t_2)$ and manage to spread them over the edges of G. We start with the node t_1 and consider the shortest path P from kW to t_1 in G. The amount of flow that can be incremented is the minimum between f and the amount of flow that t_1 can receive. Then, we increment the amount of flow on the extended path in G, which connects the source to P and connects P to the sink.

If the capacity of t_1 is reached, we decrement f by the amount of flow which was consumed and we move to the next node in the interval. Now, there remains f units of flow for the nodes within the interval $[t_1 + 1, t_2)$. By repeating the same instruction for the rest of the nodes in $[t_1, t_2)$ and for every edge in G' that carries a positive amount of flow, we obtain the flow Y. It is guaranteed that all the flow can be pushed to the nodes in $[t_1, t_2)$ as the sum of the capacities of the edges that conenct a node in $[t_1, t_2)$ to the sink in G is equal to the capacity of the edges between $[t_1, t_2)$ and the sink in the G'.

Furthermore, the flow Y satisfies the capacities since the capacities on the edges adjacent to the source in G are the same as those in G'. Moreover, the capacities were respected for the nodes adjacent to the sink. The cost of Y is the same as Y' since the paths on which the flow is pushed in Y have the same cost as the edges in the compressed graph.

We prove that Y is optimal, i.e. it is a min-cost flow. Each unit of flow in a min-cost flow in G leaves from the source to a node kW and necessarily traverses along the shortest path going to a node t and then reaches the sink. Note that the edges on the shortest path have unlimited capacities. The question is therefore on which shortest path does each unit of flow travel? The answer is the shortest path that corresponds to the edge from the compressed graph on which the flow travels. □

In what follows, RG and RG', stand for the residual graph of the scheduling graph G and the residual compressed graph G'.

Lemma 8. Let t be a node in the residual scheduling graph RG and $[t_i, t_{i+1})$, such that $t_i \leq t < t_{i+1}$, be a node in the residual compressed scheduling graph. The distance between node kW and t in RG is equal to the distance between kW and $[t_i, t_{i+1})$ in RG'.

Proof. We show that for any path P' in the residual compressed graph RG', there is a path P in the residual graph RG that has the same cost. From Lemma 6, we know that for a flow in the compressed graph G', there is an equivalent flow in the original graph G. Consider a path P' from a node kW to an interval node $[t_i, t_{i+1})$. By construction of the compressed graph, for each edge of this path corresponds a path of equal cost in the residual graph RG'. Consequently, there is a path in G' that goes from node kW to any node $t \in [t_i, t_{i+1})$ with the same cost as the path going from kW to $[t_i, t_{i+1})$ in G.

Consider a path P in the residual graph RG going from a node k_1W to a node t. Suppose that this path contains exactly one edge in RG that is not in G. We denote this edge (a,b) and the path P can be decomposed as follows: $k_1W \rightsquigarrow a \to b \rightsquigarrow t$. The edge (a,b) appears in the residual graph RG because there is a positive amount of flow circulating on a shortest path $S : k_2W \rightsquigarrow b \to a \rightsquigarrow u$ to which the reversed edge (b,a) belongs. Let Q be the following path in the residual graph RG: $k_1W \rightsquigarrow u \rightsquigarrow a \to b \rightsquigarrow k_2W \rightsquigarrow t$. Let l be the function that evaluates the cost of a path. We prove that Q has a cost that is no more than P and that it has an equivalent in the residual compressed graph RG'.

$$\begin{aligned} l(Q) &= l(k_1W \rightsquigarrow u \rightsquigarrow a \to b \rightsquigarrow k_2W \rightsquigarrow t) \\ &\leq l(k_1W \rightsquigarrow a \rightsquigarrow u \rightsquigarrow a \to b \rightsquigarrow k_2W \rightsquigarrow t) \end{aligned}$$

In the residual graph, the paths $a \rightsquigarrow u$ and $u \rightsquigarrow a$ have opposite costs, hence $l(a \rightsquigarrow u \rightsquigarrow a) = 0$.

$$\begin{aligned} &= l(k_1W \rightsquigarrow a \to b \rightsquigarrow k_2W \rightsquigarrow t) \\ &\leq l(k_1W \rightsquigarrow a \to b \rightsquigarrow k_2W \rightsquigarrow b \rightsquigarrow t) \end{aligned}$$

In the residual graph, the paths $b \rightsquigarrow k_2W$ and $k_2W \rightsquigarrow b$ have opposite costs, hence $l(b \rightsquigarrow k_2W \rightsquigarrow b)$.

$$\begin{aligned} &= l(k_1W \rightsquigarrow a \to b \rightsquigarrow t) \\ &= l(P) \end{aligned}$$

The path Q has an equivalent in the residual compressed graph RG'. Indeed, the subpaths $k_1W \rightsquigarrow u$ and $k_2W \rightsquigarrow t$ are edges in RG' whose cost is given by the shortest paths in G. The path $u \rightsquigarrow a \to b \rightsquigarrow k_2W$ is the reverse of path S. Since S is an edge in G' and there is a flow circulating on S, the reverse of S also appears in RG'. Consequently, the path P can be transformed into path Q that has an equivalent in the compressed residual graph. If P contains more than one edge that belongs to RG but not G, then the transformation can be applied multiple times.

Since a path in RG has an equivalent path whose cost is not greater in RG' and vice-versa, we conclude that a node kW is at equal distance from all the other nodes in either graph. □

Notice that the above lemma implies that after computing the min-cost flow in the compressed graph, one sets the value for x_t to the shortest distance between an arbitrary but fixed node kW to the interval node that contains t.

Let $H = o_{\mathcal{I}} - \mathrm{est}_{\mathcal{I}}$ be the horizon, we need $\Theta(\frac{H}{W})$ calls to the algorithm in [48] to build the compressed graph in $O(\frac{H}{W}n^2 \min(1, \frac{p}{m}))$ time. As in Section 6.4.1, the successive shortest paths technique, with Goldberg's algorithm [32], computes the maximum flow. The compressed graph has $|V(G')| \in O(\frac{H}{W} + n^2)$ nodes, $|E(G')| \in O(\frac{H}{W}n^2)$ edges, a maximum absolute cost of $Q \in O(n)$, and a maximum capacity of $U = \Delta c = C(W-1) - c(0)$. Computing the values for x_t requires an

additional execution of Goldberg's algorithm on the compressed graph. The final running time complexity is $O\left(\left(\left(\frac{H}{W}\right)^{2.5} + n^5\right) \Delta c \log(n)\right)$ which is faster than the algorithm presented in the previous sections when the number of periods is small, i.e. when $\frac{H}{W}$ is bounded. In practice, there are fewer periods than tasks: $\frac{H}{W} < n$.

6.5 Minimizing maximum lateness

Consider $P \mid \mathrm{est}_j; p_j = p; \mathrm{lct}_j, \bar{d}_j \mid L_{\max}$ which is the case that the tasks have due dates $\bar{d}_i$ and deadlines lct_i. One wants to minimize the maximum lateness while ensuring that tasks complete before their deadlines. To test whether there exists a schedule with maximum lateness L, one changes the deadline of all task i for $\min(\mathrm{lct}_i, \bar{d}_i + L)$. If there exists a valid schedule with this modification, then there exists a schedule with maximum lateness at most L in the original problem. Since the maximum lateness is bounded by $0 \leq L \leq \lceil \frac{np}{m} \rceil$, a well known technique consists of using the binary search that calls at most $\log(\lceil \frac{np}{m} \rceil)$ times the algorithm in [48] and achieves a running time complexity of $O(\log(\frac{np}{m}) n^2 \min(1, \frac{p}{m}))$.

6.6 Conclusion

We studied variants of the problem of non-preemptive scheduling of tasks with equal processing times on multiple machines. We proved that the objective of minimizing costs per task per time makes the problem NP-hard. We presented a polynomial time algorithm for objective functions which are monotonic with respect to the time. We also achieved a pseudo-polynomial algorithm for when the objective function is periodic and increasing within the periods. Furthermore, we generalized the problem to the case that the number of machines fluctuate through time.

Conclusion

Scheduling is a decision-making process that is concerned with the assignment of execution times to the activities. This work dealt with deterministic scheduling problems in the presence of scarce resources which must be allocated to the activities over time. The limitation on the availability of resources in this context as well as the presence of technological precedence constraints causes conflicts between concurrent scheduling of the activities over resources which makes the paradigm complex and challenging. Although branch and bound or integer programming methodologies can give rise to optimal solutions for problems whose data are relatively small, the increasing size of the problem yields quite a complex problem which requires exponential effort to solve. Therefore, it is essential to investigate more efficient methods to solve scheduling problems in large scales. The generic form of the scheduling problems considered in this thesis is represented as an instance of a constraint satisfaction problem (CSP) in which there is a set of variables, each of which associated to a set of possible values (domains), and a set of constraints interrelating the variables. An assignment of values to the variable, so that all the constraints are satisfied leads to a solution for CSP. The major objective of this work was to investigate the application of constraint programming for industrial scheduling problems. The need in practical applications in the industry is to take into consideration distinct properties of the industrial scheduling problems. The focus of this thesis was to fulfill this need by viewing the constraint-based scheduling problems from different standpoints. Since CSPs are NP-hard to solve in general, so are the scheduling problems. Nonetheless, there are specializations to the scheduling problem that can be solved in polynomial time. We aimed at developing effective solutions and designing filtering algorithms to find optimal schedules or to shrink the search space for scheduling problems in a variety of contexts.

Chapter 1 surveyed the basic concepts of constraint programming so long as they are relevant to the contents of this thesis.

In chapter 2 we introduced the definitions and notations arising in the context of resource constrain scheduling in detail. Furthermore, the global constraint that can be interpreted as special cases of the scheduling problems are introduced. Finally, we described a particular scheduling problem in which all the tasks have the same processing time and must be executed over multiple machines without preemption.

For the classical scheduling problems, there have been tremendous efforts in order to reduce the size

of the search space over the past few decades. These progresses include devising filtering algorithms which are invoked by the process of constraint propagation. To provide a theoretical foundation for the constraint propagation approach, chapter 3 reviewed different types of such filtering algorithms for the CUMULATIVE and DISJUNCTIVE constraints. These algorithms are not all used in the state of the art schedulers including Choco, Gecode or IBM ILOG CP Optimizer. Rather, a combination of some of these filtering techniques, such as Time-Tabling, Edge-Finding and Overload Checking are implemented. This is due to the fact that each one of these techniques rules out its own type of inconsistencies. Time-Tabling is sufficient to guarantee that there is no unfeasible solution. Playing an important role in these solvers, Edge-Finding performs a strong filtering based on the precedences among the tasks and Overload Checking must be integrated with the Edge-Finding algorithm to ensure that the workload of the tasks does not exceed the available energy within the window in which they execute.

The first contribution of this thesis, presented in chapter 4, focused on filtering algorithms for the DISJUNCTIVE constraint. We elaborated a new data structure, called time line, which provides constant time operations for scheduling the tasks and retrieving the earliest completion time of a set of tasks. This feature outperforms the $\Theta-$tree data structure which provides the same operations in $\log(n)$ time. We took advantage of time line data structure to present new filtering algorithms for Detectable Precedences for the DISJUNCTIVE constraint and Overload Checking for the CUMULATIVE constraint. We also used a Union-Find data structure to present an algorithm for Time-Tabling. These algorithms all have a linear running time complexity in the number of tasks. The new algorithms outperform the best algorithms known so far. The procedure was evaluated on large test sets of job shop and open shop benchmark problem instances and the influence of our techniques were analysed. We proved that sorting the tasks with the insertion sort can make the algorithms even faster. The experiments, which were obtained by implementing the insertion sort for sorting the tasks and counting the number of backtracks occurred during the traversal of the same search tree in the same order, have demonstrated the effectiveness and efficiency of our approach. The results show that as the size of the input gets larger, the impact of our methods becomes more significant, as the ratios of the backtrack numbers grow. Moreover, we showed that the time line structure is powerful enough to efficiently solve simple scheduling problems of minimizing maximum lateness and minimizing maximum lateness.

The second contribution of this thesis, presented in chapter 5, considered the scheduling problems in robust contexts, where at most r out of n tasks can be delayed while maintaining the schedule valid. We adapted the state of the art algorithms for Overload Checking and Edge-Finding by presenting the robust versions of these rules and extending the $\Theta-$tree and $\Theta - \Lambda-$tree data structures to propose new algorithms. The experimental results demonstrated a stronger filtering when our algorithms are combined with Time-Tabling when the number of delayed tasks is fixed to $r = 1$. Moreover, the algorithms run in much less computation times for many instances of the benchmark. The algorithms presented for Overload Checking and Edge-Finding can also be viewed as a general case of their counterparts for CUMULATIVE constraint. With such an approach, the Edge-Finding also improves

some inefficiencies of the algorithm in [84].

In chapter 6, which corresponds to the third contribution of this thesis, we studied variants of the problem of non-preemptive scheduling of tasks with equal processing times on multiple machines. This is a class of particular scheduling problems that can be solved in polynomial time. We generalized the problem to the case that the number of machines fluctuate through the time and adapted the weights on the scheduling graph to achieve an efficient algorithm in polynomial time. We also considered the problem with different objective functions and presented polynomial time algorithms. We considered the objective of minimizing costs per task per time, which describes the case that the cost of executing the tasks is a function of the tasks and the time. We showed that this problem is NP-complete. From that result, we focused on the special case for the objective function of minimizing costs per time, for which the cost only depends on the time of executing the tasks. We provided a polynomial time reasoning for this problem. Furthermore, we interpreted the scheduling graph in terms of a min-cost max-flow network. We took advantage of that to consider the periodic objective functions which are increasing through the periods. We introduced a compressed graph from the scheduling graph. We proved that the compressed graph maps every min-cost flow to a min-cost flow in the scheduling graph and the shortest path between the nodes can be interpreted in an equivalent manner for both graphs. By taking advantage of these properties we came up with a pseudo-polynomial time algorithms in the number of periods. Finally, we mentioned how polynomial time algorithms can be adapted with the objective of minimizing maximum lateness.

The work presented in this thesis provides better tools to solve industrial problems. Producing robust schedules is a problem that becomes more in demand by the industry. While the results in chapter 5 directly address this issue, it is interesting to see that our other contributions are also important with this respect. Indeed, a robust system can also mean a system that is able to schedule efficiently the activities upon the failure of a resource. We presented in Chapter 6 about equal processing times an efficient algorithm that can produce schedules when the number of resources fluctuates, i.e. when some machines become nonoperational. Moreover, the ability to efficiently reschedule the activities passes by faster schedulers which is what is addressed in Chapter 4

The algorithms that we presented in this thesis aimed to be compatible with complex optimization criteria encountered in the industry. Chapter 6 tackled diverse optimization criteria that are not necessarily the minimization of the makespan. These algorithms can be used to test the satisfiability of scheduling constraints that are adapted to these objective functions. Note also that the filtering algorithms presented in Chapters 4 and 5 can be used in conjunction of any optimization criteria that can be handled by a constraint solver. The level of robustness of a schedule can itself become an optimization criteria where the number of tasks r that can be delayed can be maximized by the solver. The algorithms we presented could, in future work, be adapted to compute an upper bound on r and therefore provide a complete branch-and-bound scheme for the maximization of the robustness.

It is an open question whether the time line could be used to design an Edge-Finding for the disjunctive

scheduling. It is an open question whether the time line could achieve the filtering of Detectable Precedences for the FLEXC constraint in the disjunctive case. Designing robust filtering algorithms for the alternative techniques that we considered in chapter 3, such as Extended-Edge-Finding and Not-First/Not-Last is the future work for robust scheduling problems. It is also our interest to prove that the objective of minimizing costs per time can be implemented for ANY function in polynomial time.

Table 6.1 summarizes all of the contributions that were explained in this thesis.

Algorithm	State before	State of the art	Conference/Journal
Overload Checking (CUMULATIVE)	$O(n\log(n))$ (Wolf and Schrader [91])	$O(n)$ (Fahimi, Quimper [26])	Published in AAAI 2014
Detectable Precedences (DISJUNCTIVE)	$O(n\log(n))$ (Vilím [87])	$O(n)$ (Fahimi, Quimper [26])	Published in AAAI 2014
Time-Tabling (DISJUNCTIVE)	$O(n)$ (Fahimi, Quimper [26])	$O(n)$ (Gay et. al. [30])	Published in AAAI 2014
Overload Checking (Robust CUMULATIVE)	NA	$O(r^2 n\log(n))$ (Fahimi, Quimper)	To be submitted to *Constraints* journal
Edge-Finding (Robust CUMULATIVE)	NA	$O(r^2 n\log(n))$ (Fahimi, Quimper)	To be submitted to *Constraints* journal
Variety of machines	NA	$O(\min(1, \frac{p}{m})(\|T\| + n)n)$ (Fahimi, Quimper [27])	Published in COCOA 2015
Minimizing $\sum_{i,t} w_i(S_i)$	NA	NP-Hard (Fahimi, Quimper [27])	Published in COCOA 2015
Minimizing $\sum_t w(S_i)$ for monotonic $w(S_i)$	NA	$O(\min(1,p)n^2)$ (Fahimi, Quimper [27])	Published in COCOA 2015
Scheduling and network flows	NA	$O((H - p + n^2)(H)^{3/2}\Delta c\log n)$(Fahimi, Quimper [27])	Published in COCOA 2015
Minimizing $\sum_t w(S_i)$ for periodic $w(S_i)$	NA	$O\left(\left(\left(\frac{H}{W}\right)^{2.5} + n^5\right)\Delta c\log(n)\right)$ (Fahimi, Quimper [27])	Published in COCOA 2015
Minimizing $L_{\max}$	NA	$O(\log(\frac{np}{m})n^2\min(1, \frac{p}{m}))$ (Fahimi, Quimper [27])	Published in COCOA 2015

Table 6.1 – Summary of the contributions mentioned in this thesis. NA stands for no previously known algorithms.

Bibliography

[1] Abderrahmane Aggoun and Nicolas Beldiceanu. Extending chip in order to solve complex scheduling and placement problems. *Mathematical and computer modelling*, 17(7):57–73, 1993.

[2] R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice hall, 1993.

[3] K. Artiouchine and P. Baptiste. Inter-distance constraint: An extension of the all-different constraint for scheduling equal length jobs. In *Proc. of the 11th Int. Conf. on Principles and Practice of Constraint Programming*, pages 62–76, 2005.

[4] Konstantin Artiouchine and Philippe Baptiste. Inter-distance constraint: An extension of the all-different constraint for scheduling equal length jobs. In *Principles and Practice of Constraint Programming-CP 2005*, pages 62–76. Springer, 2005.

[5] Konstantin Artiouchine and Philippe Baptiste. Arc-b-consistency of the inter-distance constraint. *Constraints*, 12(1):3–19, 2007.

[6] Konstantin Artiouchine, Philippe Baptiste, and Christoph Dürr. Runway scheduling with holding loop. *European Journal of Operational Research*, 2007.

[7] Michel L Balinski and Richard E Quandt. On an integer program for a delivery problem. *Operations Research*, 12(2):300–304, 1964.

[8] P. Baptiste. Scheduling equal-length jobs on identical parallel machines. *Discrete Applied Mathematics*, 103(1):21–32, 2000.

[9] Philippe Baptiste and Claude Le Pape. Edge-finding constraint propagation algorithms for disjunctive and cumulative scheduling. In *Proceedings of the fifteenth workshop of the UK planning special interest group*, volume 335, pages 339–345. Citeseer, 1996.

[10] Philippe Baptiste and Claude Le Pape. Constraint propagation and decomposition techniques for highly disjunctive and highly cumulative project scheduling problems. *Constraints*, 5(1-2):119–139, 2000.

[11] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-based scheduling: applying constraint programming to scheduling problems*, volume 39. Springer Science & Business Media, 2012.

[12] N. Beldiceanu and M. Carlsson. A new multi-resource cumulatives constraint with negative heights. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming (CP 2002)*, pages 63–79, 2002.

[13] Nicolas Bonifas. A o (n2 log (n)) propagation for the energy reasoning. In *Conference Paper, Roadef*, 2016.

[14] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. Boosting systematic search by weighting constraints. In *ECAI*, volume 16, page 146, 2004.

[15] Jacques Carlier. The one-machine sequencing problem. *European Journal of Operational Research*, 11(1):42–47, 1982.

[16] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[17] Thomas H Cormen. *Introduction to algorithms*. MIT press, 2009.

[18] Romuald Debruyne and Christian Bessiere. Some practicable filtering techniques for the constraint satisfaction problem. In *In Proceedings of IJCAI'97*. Citeseer, 1997.

[19] Romuald Debruyne and Christian Bessiére. Domain filtering consistencies. *J. Artif. Intell. Res.(JAIR)*, 14:205–230, 2001.

[20] Alban Derrien and Thierry Petit. A new characterization of relevant intervals for energetic reasoning. In *Principles and Practice of Constraint Programming*, pages 289–297. Springer, 2014.

[21] Alban Derrien, Thierry Petit, and Stéphane Zampelli. A declarative paradigm for robust cumulative scheduling. In *Principles and Practice of Constraint Programming*, pages 298–306. Springer, 2014.

[22] Mehmet Dincbas, Pascal Van Hentenryck, Helmut Simonis, Abderrahmane Aggoun, Thomas Graf, Françoise Berthier, et al. The constraint logic programming language chip. In *FGCS*, volume 88, pages 693–702, 1988.

[23] Ulrich Dorndorf. *Project Scheduling with Time Windows: From Theory to Applications; with 17 Tables*. Springer Science & Business Media, 2002.

[24] C. Dürr and M. Hurand. Finding total unimodularity in optimization problems solved by linear programs. *Algorithmica*, 2009. DOI 10.1007/s00453-009-9310-7.

[25] Jacques Erschler. *Analyse sous contraintes et aide à la décision pour certains problèmes d'ordonnancement*. PhD thesis, 1976.

[26] Hamed Fahimi and Claude-Guy Quimper. Linear-time filtering algorithms for the disjunctive constraint. In *AAAI*, pages 2637–2643, 2014.

[27] Hamed Fahimi and Claude-Guy Quimper. Variants of multi-resource scheduling problems with equal processing times. In *Combinatorial Optimization and Applications*, pages 82–97. Springer, 2015.

[28] H. N. Gabow and R. E. Tarjan. A linear-time algorithm for a special case of disjoint set union. In *Proceedings of the 15th annual ACM symposium on Theory of computing*, pages 246–251, 1983.

[29] Michael R Garey and David S Johnson. A guide to the theory of np-completeness. *WH Freemann, New York*, 1979.

[30] Steven Gay, Renaud Hartert, and Pierre Schaus. Simple and scalable time-table filtering for the cumulative constraint. In *Principles and Practice of Constraint Programming*, pages 149–157. Springer, 2015.

[31] Gecode Team. Gecode: Generic constraint development environment, 2006. Available from `http://www.gecode.org`.

[32] A. V. Goldberg. Scaling algorithms for the shortest paths problem. *SIAM Journal on Computing*, 24(3):494–504, 1995.

[33] Ronald L Graham, Eugene L Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5:287–326, 1979.

[34] R. Harter. The minimum on a sliding window algorithm. Usenet article, 2001. http://richardhartersworld.com/cri/2001/slidingmin.html.

[35] Pascal Van Hentenryck and Laurent Michel. *Constraint-based local search*. The MIT Press, 2009.

[36] John N Hooker. Planning and scheduling to minimize tardiness. In *Principles and Practice of Constraint Programming-CP 2005*, pages 314–327. Springer, 2005.

[37] John N Hooker. *Integrated methods for optimization*, volume 100. Springer Science & Business Media, 2007.

[38] John N Hooker and Greger Ottosson. Logic-based benders decomposition. *Mathematical Programming*, 96(1):33–60, 2003.

[39] W. A. Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21(1):177–185, 1974.

[40] Roger Kameugne and Laure Pauline Fotso. A cumulative not-first/not-last filtering algorithm in o (n 2log (n)). *Indian Journal of Pure and Applied Mathematics*, 44(1):95–115, 2013.

[41] Roger Kameugne, Laure Pauline Fotso, and Joseph Scott. A quadratic extended edge-finding filtering algorithm for cumulative resource constraints. *International Journal of Planning and Scheduling*, 1(4):264–284, 2013.

[42] Roger Kameugne, Laure Pauline Fotso, Joseph Scott, and Youcheu Ngo-Kateu. A quadratic edge-finding filtering algorithm for cumulative resource constraints. *Constraints*, 19(3):243–269, 2014.

[43] Abdelkader Lahrichi. Scheduling-the notions of hump, compulsory parts and their use in cumulative problems. *comptes rendus de l academie des sciences serie i-mathematique*, 294(6):209–211, 1982.

[44] Michel Leconte. A bounds-based reduction scheme for constraints of difference. In *Proceedings of the Constraint-96 International Workshop on Constraint-Based Reasoning*, pages 19–28, 1996.

[45] A. Letort, N. Beldiceanu, and M. Carlsson. A scalable sweep algorithm for the cumulative constraint. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP 2012)*, pages 439–454, 2012.

[46] J. Y-T. Leung. *Handbook of scheduling: algorithms, models, and performance analysis*. CRC Press, 2004.

[47] W. Lipski Jr and F. P. Preparata. Efficient algorithms for finding maximum matchings in convex bipartite graphs and related problems. *Acta Informatica*, 15(4):329–346, 1981.

[48] A. López-Ortiz and C.-G. Quimper. A fast algorithm for multi-machine scheduling problems with jobs of equal processing times. In *Proc. of the 28th Int. Symposium on Theoretical Aspects of Computer Science (STACS'11)*, pages 380–391, 2011.

[49] A. López-Ortiz, C.-G. Quimper, J. Tromp, and P. van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI-03)*, pages 245–250, 2003.

[50] Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter Van Beek. A fast and simple algorithm for bounds consistency of the alldifferent constraint. In *IJCAI*, volume 3, pages 245–250, 2003.

[51] Alan K Mackworth. Consistency in networks of relations. *Artificial intelligence*, 8(1):99–118, 1977.

[52] Kurt Mehlhorn and Sven Thiel. Faster algorithms for bound-consistency of the sortedness and the alldifferent constraint. In *Principles and Practice of Constraint Programming–CP 2000*, pages 306–319. Springer, 2000.

[53] L. Mercier and P. Van Hentenryck. Edge finding for cumulative scheduling. *INFORMS Journal on Computing*, 20(1):143–153, 2008.

[54] Ugo Montanari. Networks of constraints: Fundamental properties and applications to picture processing. *Information sciences*, 7:95–132, 1974.

[55] WPM Wim Nuijten. *Time and resource constrained scheduling: a constraint satisfaction approach*. PhD thesis, Technische Universiteit Eindhoven, 1994.

[56] P. Ouellet and C.-G. Quimper. Time-table-extended-edge-finding for the cumulative constraint. In *Proceedings of the 19th International Conference on Principles and Practice of Constraint Programming (CP 2013)*, pages 562–577, 2013.

[57] Pierre Ouellet and Claude-Guy Quimper. The multi-inter-distance constraint. In *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, volume 22, page 629, 2011.

[58] Pierre Ouellet and Claude-Guy Quimper. Time-table extended-edge-finding for the cumulative constraint. In *Principles and Practice of Constraint Programming*, pages 562–577. Springer, 2013.

[59] Claude Le Pape. Implementation of resource constraints in ilog schedule: a library for the development of constraint-based scheduling systems. *Intelligent Systems Engineering*, 3(2):55–66, 1994.

[60] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC, INRIA Rennes, LINA CNRS UMR 6241, COSLING S.A.S., 2015.

[61] Jean-François Puget. A fast algorithm for the bound consistency of alldiff constraints. In *AAAI/IAAI*, pages 359–366, 1998.

[62] Claude-Guy Quimper. *Efficient propagators for global constraints*. PhD thesis, University of Waterloo, 2006.

[63] Claude-Guy Quimper, Alejandro López-Ortiz, and Gilles Pesant. A quadratic propagator for the inter-distance constraint. In *PROCEEDINGS OF THE NATIONAL CONFERENCE ON ARTIFICIAL INTELLIGENCE*, volume 21, page 123. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2006.

[64] Claude-Guy Quimper, Peter Van Beek, Alejandro López-Ortiz, Alexander Golynski, and Sayyed Bashir Sadjad. An efficient bounds consistency algorithm for the global cardinality constraint. In *Principles and Practice of Constraint Programming–CP 2003*, pages 600–614. Springer, 2003.

[65] Philippe Refalo. Impact-based search strategies for constraint programming. In *Principles and Practice of Constraint Programming–CP 2004*, pages 557–571. Springer, 2004.

[66] Jean-Charles Régin. A filtering algorithm for constraints of difference in csps. In *AAAI*, volume 94, pages 362–367, 1994.

[67] Jean-Charles Régin. Generalized arc consistency for global cardinality constraint. In *Proceedings of the thirteenth national conference on Artificial intelligence-Volume 1*, pages 209–215. AAAI Press, 1996.

[68] Jean-Charles Régin. The global minimum distance constraint. Technical report, Technical report, ILOG, 1997.

[69] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of constraint programming*. Elsevier, 2006.

[70] Jürgen Sauer. Knowledge-based systems techniques and applications in scheduling. *Knowledge-Based Systems Techniques and Applications, TL Leondes, ed., San Diego, Academic Press*, 1999.

[71] Andreas Schutt, Thibaut Feydy, and Peter J Stuckey. Explaining time-table-edge-finding propagation for the cumulative resource constraint. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 234–250. Springer, 2013.

[72] Andreas Schutt, Armin Wolf, and Gunnar Schrader. Not-first and not-last detection for cumulative scheduling in {\ cal O}(n^ 3\ log n). In *Declarative Programming for Knowledge Management*, pages 66–80. Springer, 2005.

[73] Joseph Scott. Filtering algorithms for discrete cumulative resources. 2010.

[74] B. Simons. Multiprocessor scheduling of unit-time jobs with arbitrary release times and deadlines. *SIAM Journal on Computing*, 12(2):294–299, 1983.

[75] B. B Simons and M. K. Warmuth. A fast algorithm for multiprocessor scheduling of unit-length jobs. *SIAM Journal on Computing*, 18(4):690–710, 1989.

[76] Barbara Simons. A fast algorithm for single processor scheduling. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 246–252. IEEE, 1978.

[77] SF Smith. Reactive scheduling systems, intelligent scheduling systems, de brown and wt scherer, 1995.

[78] E. Taillard. Benchmarks for basic scheduling problems. *European Journal Opererational Research*, 64(2):278–285, 1993.

[79] Alexander Tesch. A nearly exact propagation algorithm for energetic reasoning in o (nˆ 2 log n). 2016.

[80] Edward Tsang. *Foundations of Constraint Satisfaction: The Classic Text*. BoD–Books on Demand, 2014.

[81] Pascal Van Hentenryck and Vijay Saraswat. Constraint programming: Strategic directions. *Constraints*, 2(1):7–33, 1997.

[82] P. Vilím. $O(n \log n)$ filtering algorithms for unary resource constraint. In *Proceedings of the 1st International conference on Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimisation Problems (CPAIOR 2004)*, pages 335–347, 2004.

[83] P. Vilím. *Global Constraints in Scheduling*. PhD thesis, Charles University in Prague, 2007.

[84] P. Vilím. Edge finding filtering algorithm for discrete cumulative resources in o(kn log n). In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming*, pages 802–816, 2009.

[85] P. Vilím, R. Barták, and O. Čepek. Unary resource constraint with optional activities. In *Proceedings of the 10th International Conference on Principles and Practice of Constraint Programming*, pages 62–76, 2004.

[86] Petr Vilím. Batch processing with sequence dependent setup times. In *International Conference on Principles and Practice of Constraint Programming*, pages 764–764. Springer, 2002.

[87] Petr Vilím. O (nlog n) filtering algorithms for unary resource constraint. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 335–347. Springer, 2004.

[88] Petr Vilım. *Global constraints in scheduling*. PhD thesis, PhD thesis, Charles University in Prague, Faculty of Mathematics and Physics, Department of Theoretical Computer Science and Mathematical Logic, KTIML MFF, Universita Karlova, Malostranské námestı 2/25, 118 00 Praha 1, Czech Republic, 2007.

[89] Petr Vilím. Max energy filtering algorithm for discrete cumulative resources. In *Integration of AI and OR techniques in constraint programming for combinatorial optimization problems*, pages 294–308. Springer, 2009.

[90] Petr Vilím. Timetable edge finding filtering algorithm for discrete cumulative resources. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, pages 230–245. Springer, 2011.

[91] Armin Wolf and Gunnar Schrader. {\ cal O}(n\ log n) overload checking for the cumulative constraint and its application. In *Declarative Programming for Knowledge Management*, pages 88–101. Springer, 2005.

[92] L. A. Wolsey and G. L Nemhauser. *Integer and combinatorial optimization*. John Wiley & Sons, 2014.

[93] Monte Zweben and Mark Fox. *Intelligent scheduling*. Morgan Kaufmann Publishers Inc., 1994.