# "Constraint programming algorithms and models for scheduling applications"

Dejemeppe, Cyrille

**Abstract**

Time-related optimization problems are very hard to solve. Scheduling covers a subcategory of such problems where the goal is to place events in time. The discretization of time triggers a rapid growth of possible placements, leading to large search spaces. While scheduling problems were already computationally solved two decades ago, large real-world sized instances seemed to be out of reach back then. The recent gain of computational power brought by modern computers is not large enough to counter the combinatorial explosion of scheduling problems. However, with the recent developments of these two last decades, new promising resolution techniques have emerged. One of these techniques that will be at the core of this thesis is Constraint Programming (CP). This paradigm allows to express declaratively discrete constrained optimization problems. This thesis achieves two main goals. First, we design new abstractions and techniques to enrich the set of tools CP can use to solve Schedul...

Document type : *Thèse (Dissertation)*

# CONSTRAINT PROGRAMMING ALGORITHMS AND MODELS FOR SCHEDULING APPLICATIONS

CYRILLE DEJEMEPPE

*Thesis submitted in partial fulfillment of the requirements for the degree of Doctor of Applied Science in Engineering*

August 2016

Institute of Information and Communication Technologies,
Electronics and Applied Mathematics
Louvain School of Engineering
Louvain-la-Neuve
Belgium


**Thesis Committee**

| | |
|---|---|
| Yves Deville (director) | UCLouvain, Belgium |
| Pierre Schaus (director) | UCLouvain, Belgium |
| Nicolas Beldiceanu | École des mines de Nantes, France |
| Jean-Noël Monette | Tacton Systems AB, Sweden |
| Daniele Catanzaro | UCLouvain, Belgium |
| Charles Pecheur (president) | UCLouvain, Belgium |

# ABSTRACT

Time-related optimization problems are very hard to solve. Scheduling covers a subcategory of such problems where the goal is to place events in time. The discretization of time triggers a rapid growth of possible placements, leading to large search spaces. While scheduling problems were already computationally solved two decades ago, large real-world sized instances seemed to be out of reach back then. The recent gain of computational power brought by modern computers is not large enough to counter the combinatorial explosion of scheduling problems. However, with the recent developments of these two last decades, new promising resolution techniques have emerged. One of these techniques that will be at the core of this thesis is Constraint Programming (CP). This paradigm allows to express declaratively discrete constrained optimization problems.

This thesis achieves two main goals. First, we design new abstractions and techniques to enrich the set of tools CP can use to solve Scheduling problems. Second, we prove that CP is able to solve large instances of real-world scheduling problems in short amounts of time.

Several new scheduling abstractions for CP are introduced in this thesis. The two most important ones are the conception of two new propagation procedures useful for time-related problems. Propagation is a mechanism of CP that allows to remove parts of the search space that are provably unfeasible, i.e., that do not contain any feasible solutions. The first propagation procedure introduced performs Forward-Checking propagation for the Nested Global Cardinality Constraint

(Nested GCC). This constraint allows to bound the number of occurrences of values on nested ranges of variables. The second propagation procedure is designed for a new constraint: the unary resource with transition times. This constraint forbids a group of activities to overlap in time and imposes a minimal delay between each pair of activities from this group.

We apply CP scheduling resolution techniques on four industrial problems. These four problems take place in two main sectors of activity: medical treatment centers and industrial production. The tree first problems attempt at scheduling the steps followed by patients in radiotherapy treatment centers. The last problem aims at reducing the energy costs of industrial sites by shifting their most energy-demanding production tasks when electricity prices are the lowest. These four problems have been solved on either realistic generated instances or on historical instances. By coupling CP with Large Neighborhood Search, a diversification strategy, we have been able to provide high quality solutions to these four problems.

# ACKNOWLEDGEMENTS

This thesis is not mine alone, many people have contributed to it. First of all, I would like to thank my two *sensei*: Yves and Pierre. Yves is a great advisor, his counseling and guidance reveal all his experience. I have learned a lot from him: how to write good papers, the art of teaching, etc. We also had a bunch of very interesting philosophic conversations and his human side has never ceased to amaze me. Pierre has also largely contributed to this thesis. He has always been great at feeding my thirst for knowledge and challenges with many interesting problems, even before this thesis had even started. We have written great papers together and we still have many plans (too much, actually) for future ones. I have also learned how to write efficient code in a large open source project, OscaR, under his supervision. It was a real delight working with both of you, Yves and Pierre.

I would also like to thank many people from the INGI department. First of all, thank you to my *senpai*, Florence and JB. Florence has been there since before this thesis even began and her advices were always the right thing to do, either for work or for personal matters. We also had great fun together and Florence is really a great person to have in your office or at your side in conference. As for JB, I have been at more conferences sleeping in a single room with him than in my own room (and he would love to joke about it). He is also super cool and we had a lot of beers and fun together. Flo and JayBe, stay cool senpai! Many thanks go also to the other BeCool members but special mentions go to my *nakama* Renaud, Steven and Sascha. They were always there to

seriously discuss about scheduling and OscaR performances, have a drink at the cafeteria, drink beers (and Coke for Steven) or dive in a pile of grain. A huge thank you goes to Sascha with whom I have written (and am still writing) super papers. We also had great fun in many conferences and maybe will again in the near future.

I also thank all the INGI members, researchers and members of the administrative and technical staff. Special mentions go to Vanessa who is always there to crack/laugh at a good joke and Pierre (3.14r) who has been a super yoga teacher. I also thank all the members of the CORSCI and especially Stéphanie with whom I have cursed a lot on RedDaube.

I wish to thank all of the persons I have had contact with during the two Walloon region projects I have worked on. Thank you Damien and François for all the support on the MIRROR project. Thank you to all the members of N-Side for their warm welcome during the InduStore project. We had great time between work, pies for birthdays, babyfoot tournaments, minifoot matches and so on.

I would like to thank all of my friends for the various moments we have spent together. I could cite all of your names but unfortunately I have space constraints. Yes, this thesis has been often the reason for which I haven't been able to hang out with you guys. Many warm thanks go to FX who has accepted to read-proof early versions of this thesis.

To all my family, I say thank you. Brothers and sisters, thank you for your support, sometimes well disguised. Special thanks go to my parents who have always believed in me and have largely contributed to make me who I am today. Special kudos for my mom who has just achieved something far more greater than this thesis this last year.

Thank you Bill & Rebecca for the awesome musical moments spent on Radio Paradise.

I also thank all the members of my thesis jury for accepting to read my thesis. I warmly thank Jean-Noël for being there since the beginning of my thesis.

Sanji, here are a few words for you too: "Meow? Meow! Meow, meow, meeeeeow".

Finally, last but not least: Morgane. Without you, this thesis would only be the shadow of itself. I cannot count all the moments you were there to support me along this work. Let us continue to grow stronger together. You rock my world baby!

Cyrille, May 2016

# REMERCIEMENTS

Cette thèse n'a pas été écrite par mon seul, beaucoup de personnes y ont contribué. Premièrement, je voudrais remercier mes deux *sensei*: Yves et Pierre. Yves est un très bon promoteur, son encadrement et ses conseils laissent transparaître toute son expérience. Il m'a appris beaucoup de choses: comment écrire des supers articles, l'art de l'enseignement, etc. Nous avons aussi eu pas mal de conversations philosophiques très intéressantes et son côté humain n'a jamais cessé de m'impressionner. Pierre a aussi largement contribué à cette thèse. Il a toujours dédié ses efforts à entretenir ma soif de connaissance et de challenges à l'aide de nombreux problèmes intéressants, et ce bien avant que cette thèse ne débute. Nous avons écrit des super papiers ensemble et nous avons encore pas mal d'idées (trop en fait) pour d'autres qu'il faudrait encore écrire. J'ai aussi appris sous sa supervision à écrire du code efficace dans un large projet open-source, OscaR. Ce fut un réel plaisir de travail en votre compagnie, Yves et Pierre.

Je voudrais aussi remercier beaucoup de membres du département INGI. Premièrement, merci à mes *senpai*, Florence et JB. Florence est présente à mes côtés depuis bien avant cette thèse et ses conseils ont toujours été justes, que ce soit dans le cadre du travail ou sur le plan personnel. Nous avons passés de très bons moments ensemble et Florence est vraiment une personne dont la présence est agréable, que ce soit dans son bureau ou en conférence. Quant à JB, j'ai été à plus de conférences en partageant la même chambre avec lui que dans une chambre pour moi seul (et il adorerait lancer des vannes à ce sujet). Il

est aussi super cool et nous avons dégusté pas mal de bières et passé beaucoup de bons moments ensemble. Flo et JayBe, restez cools senpai! Je remercie aussi chaleureusement les autres membres BeCool et plus particulièrement mes *nakama* Renaud, Steven et Sascha. Ils ont toujours été là pour discuter sérieusement de scheduling et des performances d'OscaR, pour boire un café (plutôt plusieurs) à la cafétéria, boire des bières (et un coca pour Steven) ou plonger dans un tas de grain. Un tout grand merci à Sascha avec lequel j'ai écrit (et suis encore en train de le faire) de super papiers. Nous nous sommes aussi bien amusé dans plusieurs conférences et nous remettrons peut-être ça dans un futur proche.

Je remercie aussi tous les membres INGI, les chercheurs et les membres du staff administratif et technique. Des mentions spéciales vont à Vanessa qui est toujours là pour rigoler/lancer une bonne blague et Pierre (3.14r) qui a été un super prof de yoga. Je remercie aussi tous les membres du CORSCI et spécialement Stéphanie avec qui j'ai beaucoup pesté sur le framework RedDaube.

J'aimerais aussi remercier toutes les personnes avec qui j'ai été en contact au cours des deux projets de la région wallonne sur lesquels j'ai travaillé. Merci à Damien et François pour leur soutient sur le projet MIRROR. Merci à tous les membres de N-Side pour leur accueil chaleureux au cours du projet INduStore. Nous avons passé un bon nombre de moments mémorables entre le travail, les tartes d'anniversaire, les tournois de kicker, les matchs de minifoot etc.

Je souhaite aussi remercier tous mes amis pour les nombreux moments que nous avons passés ensemble. J'aurais voulu tous les citer mais j'ai malheureusement des contraintes d'espace. Oui, c'est bien cette thèse qui est la raison pour laquelle je n'ai pas toujours été capable de passer du temps avec vous les gars. Des remerciements chaleureux vont à FX qui a accepté de relire la version beta de ce manuscrit.

À toute ma famille, je dis merci. Chers frères et sœurs, merci pour votre soutien, parfois bien déguisé. J'adresse des remerciements spéciaux à mes parents qui ont toujours eu foi en moi et qui ont largement contribué à la personne que je suis aujourd'hui. Des kudos spéciaux sont dédiés à ma maman qui a accompli quelque chose de beaucoup plus important que cette thèse cette année.

Merci Bill & Rebecca pour les merveilleux moments musicaux que j'ai passé connecté à Radio Paradise.

Je remercie aussi tous les membres de ce jury qui ont accepté de lire ma thèse. Je remercie chaleureusement Jean-Noël qui est dans mon comité d'accompagnement depuis le début de ma thèse.

Sanji, voici quelques mots pour toi aussi: "Meow? Meow! Meow, meow, meeeeeow".

Enfin, je remercie la dernière et non des moindre: Morgane. Sans toi, cette thèse ne serait même pas l'ombre de ce qu'elle est. J'ai perdu le compte de tous les moments où tu étais à mes côtés pour me soutenir tout au long de ce travail. Ne nous arrêtons pas là et continuons à grandir ensemble. You rock my world baby!

Cyrille, Mai 2016

# CONTENTS

# INTRODUCTION

## TIME-RELATED PROBLEMS IN CONSTRAINT PROGRAMMING

Since mankind has been able to perform philosophic reasoning, *time* has always been fascinating us. Zeno of Elea (490–430 BC), a Greek philosopher, was already measuring the time needed by Achilles to catch up a tortoise. The recent advances of science are completely modifying the vision we have of this abstract concept. With the evolution of computation power brought by modern computers, we are able to solve larger and larger time-related problems.

This thesis aims at solving time-related problems in a combinatorial optimization context. The most widely explored category of problems that we investigate in this thesis is called scheduling. Scheduling groups a wide range of problems in which the aim is to determine *when* a set of events must take place. The possible placements of these events are subject to constraints. These constraints can link events together, impose that some events use a resource, etc. Most of the time, finding a feasible solution to a scheduling problem is not enough: the quality of the solution also matters. In scheduling, optimization aims at finding a possible schedule such that one or several optimization criteria are either minimized or maximized.

We have used Constraint Programming (CP) to solve scheduling problems. CP is a paradigm to solve constrained problems. This paradigm allows to formally define a problem in which constraints are declaratively stated. As such, a CP model describes the solutions but

not how to obtain them. CP models define the search space containing all the feasible solutions. For many problems, the search space is so large that, even with computer billions of time more powerful than those we have right now, it would take billions of times the age of the universe to totally browse it. Propagation is a technique of the CP paradigm aiming at removing parts of the search space that are provably unfeasible. This reduction of the search space allows to tackle several large problems that would not be solvable otherwise.

In this thesis, we propose several new abstractions, including two propagation procedures for CP. The first propagation procedure developed in this thesis deals with the Nested Global Cardinality Constraint. The Nested GCC restricts the number of occurrences of values on embedded ranges of variables. Considering a range of four variables $x_1, x_2, x_3, x_4$, it could for example impose that at least 1 variable takes the value 2 in the range $x_1, x_2, x_3$ and at most two variables take the value 1 in the range $x_1, x_2, x_3, x_4$. The second propagation procedure we develop in this thesis is for the unary resource with transition times. In scheduling problems, it constrains two events not to overlap in time and impose a minimal delay between these events. Other additional abstractions are also presented in this thesis. They are highlighted and detailed in the chapters of this thesis.

This thesis has been realized under two successive grants from the Walloon region. These grants were provided to realize research consulting for industrial projects.

For the first project called MIRROR, we have proposed and solved scheduling models to optimize the schedule of patient treatment in medical centers. Two different contexts were considered: Proton Therapy (PT) centers and Nuclear Medicine (NM) centers. Proton Therapy is a technique used in the treatment of cancer. During each PT treatment session, the tumor of a patient is exposed to a beam of accelerated protons. Successive sessions will attack the tumor, hopefully leading to a full recovery for the patient. On the other hand, Nuclear Medicine centers are used for diagnostic reasons. In NM centers, patients are injected with nuclear tracers. As these tracers decay, they emit radiations that can be captured by scanner sensors, allowing to obtain an image of the desired part of the body.

The second industrial project, called InduStore, aims at reducing energy costs of industrial sites. The prices of electricity on the European market varies hour by hour, sometimes dramatically. The aim of this project is to propose production plans for industrial sites such that

their global electricity costs are reduced. To achieve that our model attempts to schedule production processes with high energy demands when the electricity prices (forecasts) are lower.

All these industrial problems have been solved using the abstractions we introduce in this thesis. We use CP in combination with Large Neighborhood Search (LNS), a strategy exploring the neighborhood of solutions. We show in this thesis how CP + LNS is able to solve several hard industrial problems on real-world instances in relatively small amounts of time.

Grab your best reading glasses, grab a good Belgian beer, and onwards with the reading. For the glory of Science, may the Force be with you!

## CONTRIBUTIONS

The main contributions of this thesis are the following:

### THE UNARY RESOURCE WITH TRANSITION TIMES CONSTRAINT

We introduce a new global constraint: the unary resource with transition times. An efficient propagation procedure is detailed for this constraint. This procedure contains propagators running in $\mathcal{O}(n \log(n))$ where $n$ is the number of activities on the considered unary resource. This propagation procedure is compared to other state-of-the art propagators.

### AN FWC PROPAGATION PROCEDURE FOR THE NESTED GCC

We introduce a new Forward Checking propagation procedure for the Nested GCC constraint. We propose a pre-computation step strengthening the bounds for this constraint, allowing additional pruning over initial bounds. A dedicated propagator running in $\mathcal{O}(\log(p))$ is also introduced, where $p$ is the number of ranges of variables constrained by the constraint.

### RESOLUTION OF THE PROTON THERAPY PROBLEM

We propose two different models for the Proton Therapy Problem. A model is described and solved using a CP + LNS approach. We have solved this problem for large real-world instances.

### TEN WEEKS AHEAD APPOINTMENT SCHEDULE PROBLEM

We propose reactive optimization for the Ten Weeks Ahead Appointment Schedule Problem. The reactive approach is able to solve real-

world sized instances using CP + LNS in reasonably small amounts of time.

RESOLUTION OF THE NUCLEAR MEDICINE PROBLEM

We solve the multi-objective Nuclear Medicine Problem using CP + LNS. Two new propagation procedures are introduced for constraints dealing with the exponential components of the problem. These propagation procedures rely on two new views for view-based propagator derivation.

RESOLUTION OF THE PAPER PRODUCTION PLANNING PROBLEM

We solve the Paper Production Planning Problem using a CP + LNS approach. We solve this problem over historical instances and we prove the benefits brought by the optimization resolution.

Several fragments of the source code produced for this thesis are detailed in Appendix A. This thesis has also brought many contributions to the open-source OscaR solver [Osc12].

## PUBLICATIONS

The four main publications discussed in this thesis are the following.

### WORKSHOP PUBLICATIONS

[Dej13] Cyrille Dejemeppe. "Alternative Job-Shop Scheduling For Proton Therapy." In: *CP Doctoral Program 2013* (2013), p. 67.

### CONFERENCE PUBLICATIONS

[DCS15] Cyrille Dejemeppe, Sascha Van Cauwelaert, and Pierre Schaus. "The Unary Resource with Transition Times." In: *Principles and Practice of Constraint Programming*. Cork, Ireland: Springer International Publishing, 2015.

[DD14] Cyrille Dejemeppe and Yves Deville. "Continuously Degrading Resource and Interval Dependent Activity Durations in Nuclear Medicine Patient Scheduling." In: *Integration of AI and OR Techniques in Constraint Programming*. Cork, Ireland: Springer International Publishing, 2014, pp. 284–292.

[Dej+16] Cyrille Dejemeppe, Olivier Devolder, Victor Lecomte, and Pierre Schaus. "Forward-Checking Filtering for Nested Cardinality Constraints: Application to an Energy Cost-Aware Production Planning Problem for Tissue Manufacturing." In: *Integration of AI and OR Techniques in Constraint Programming*. Banff, Canada: Springer International Publishing, 2016.

In addition to those, two other papers have been written during this thesis. The first one has already been published and presented at the CPAIOR 2015 conference. It was however not in the scope of this thesis and therefore left out on purpose. The second paper has been submitted at the CP 2016 conference, but has not yet been accepted. This paper, while in the scope of this thesis, has been written during the writing of this thesis manuscript and therefore left out of it.

### CONFERENCE PUBLICATIONS

[Cau+16] Sascha Van Cauwelaert, Cyrille Dejemeppe, Jean-Noël Monette, and Pierre Schaus. "Efficient Filtering for the Unary Resource with Family-based Transition Times." In: *Principles and Practice of Constraint Programming (submitted, to be accepted)*. Springer International Publishing, 2016.

[DSD15] Cyrille Dejemeppe, Pierre Schaus, and Yves Deville. "Derivative-Free Optimization: Lifting Single-Objective to Multi-Objective Algorithm." In: *Integration of AI and OR Techniques in Constraint Programming*. Barcelona, Spain: Springer International
Publishing, 2015, pp. 124–140.

### OUTLINE

This thesis is divided in four parts. The first part contains background information about the technical aspects of the thesis. The second part introduces propagation procedures associated to constraints used to solve time-related problems in CP. The third part presents scheduling applications in the context of radiotherapy treatment centers. The fourth part describes the resolution of a production planning industrial problem.

In Part I, Chapter 1 dives into the main concepts of the Constraint Programming paradigm. It gives all the key concepts needed to understand the role and internal mechanisms of constraint propagation. Then, Chapter 2 presents Scheduling, a category of time-related problems. All the components needed to understand the various scheduling models of this thesis are described there. To conclude this part, Chapter 3 describes performance profiles in the context of propagator comparison. This technique is used throughout this thesis to compare various propagation procedures.

In Part II, Chapter 4 introduces a new propagation procedure in Forward Checking for the Nested GCC. It also introduces a pre-computation procedure aiming at strengthening bounds to achieve stronger pruning. Chapter 5 introduces the new unary resource with transition times constraint. A dedicated propagation procedure is also detailed in this chapter.

Part III begins with Chapter 6 that introduces the Proton Therapy Problem and the Ten Weeks Ahead Appointment Schedule Problem. These two problems aim at scheduling patient treatment sessions in a Proton Therapy center. Chapter 7 tackles the Nuclear Medicine Problem in which patient treatment sessions have to be scheduled in a Nuclear Medicine center.

Finally, Part IV contains a single chapter: Chapter 8. This chapter tackles the resolution of the Paper Planning Production Problem. It also tests the model and the CP + LNS resolution strategy on historical instances.

# Part I

# BACKGROUND

# 1

# CONSTRAINT PROGRAMMING

*With great power comes great responsibility.*

—Uncle Ben, *Spider-Man*

*Do, or do not. There is no 'try'.*

—Yoda, *Star Wars: Episode V - The Empire Strikes Back*

*No matter how small you start, always dream big.*

—Stephen Richards

*The more constraints one imposes, the more one frees one's self. And the arbitrariness of the constraint serves only to obtain precision of execution.*

—Igor Stravinsky

*Problems are hidden opportunities, and constraints can actually boost creativity.*

—Martin Villeneuve

As described in [WN14], *Combinatorial Optimization Problems* attempts to efficiently allocate a set of limited resources to optimize a determined objective where the concerned resources can only be divided into discrete parts. These resources may be machines, people or any other discrete quantity and the constraints on these resources restricts the possible alternatives to a finite set. Usually, the amount of possible alternatives is too large to be completely enumerated for instances of realistic size. For example, a steel production site might desire to plan its production such that its electricity costs are minimized. Combinatorial optimization problems are modeled with the help of *variables*, terms used to represent integer quantities of the problem. These variables can be *assigned* to a limited set of *values*. A *complete* assignment (or *solution*) is the assignation of all the variables to a single value. The *search space* is the set of all potential assignments of values to the variables of the problem. A *feasible solution* to the problem is a complete assignment respecting all the constraints. When dealing with an *optimization* problem, an *objective function* is used to associate a preference value (referred to as *objective value*) to each possible solution. The resolution of a combinatorial optimization problem is to find the feasible solution with the most preferred objective value, e.g., in a minimization context, this would be the feasible solution with the smallest objective value. Many optimization techniques exist and this chapter aims at describing one of them: Constraint Programming

Constraint Programming (CP) [Apto3, RVW06, Pie15] is a paradigm using constraint abstractions for solving hard combinatorial problems. Constraint Programming is declarative; it describes the search space with variables and possible values for those; then it states constraints restricting possible assignments to feasible solutions. This means that CP only defines what are the possible solutions; it does not describe *how* to obtain them. As other combinatorial optimization techniques, the aim of CP is to find feasible solutions inside the search space. A solution is said to be *feasible* if it lies within the search space and all constraints are satisfied. Constraint Programming is a potentially *complete* resolution technique. It can be used to search *all* the feasible solutions in the search space. This means that it will explore the whole search space – or at least the whole feasible region inside the search space. To avoid browsing all potential solutions, CP methodically removes unfeasible regions from the search space. This is done with the help of *propagation* procedures associated to the constraints of the problem.

CP aims at solving two different categories of problems. In the first category, one is interested in finding a single feasible solution. Such problems are called *Constraint Satisfaction Problems* (CSPs). In the second category of problems, one is interested in finding the best feasible solution according to a set of criteria. Such problems are called *Constraint Optimization Problems* (COPs). In Section 1.1, we define Constraint Satisfaction Problems; similarly, Section 1.2 defines Constraint Optimization Problems. Then, Section 1.3 describes the propagation mechanisms associated to constraints allowing to reduce the search space. Section 1.4 explains how the browsing of the search space is performed in a CP framework. Finally, Section 1.5 describes the Large Neighborhood Search metaheuristic used in combination with CP to solve hard COPs.

## 1.1 CONSTRAINT SATISFACTION PROBLEM

A Constraint Satisfaction Problem (CSP) is a representation of a combinatorial optimization problem. It is defined by a set of decisions variables, a set of domains (possible values for variables), and a set of constraints restricting assignments of values to variables. The domains associated to variables are discrete. The aim of solving a CSP is to assign a single value to each decision variable from its domain such that the constraints are satisfied.

Formally, a CSP is a triplet $\text{CSP}(X, D, C)$ where $X$ is a set of $n$ *variables*, $D$ is a set of $n$ *domains* and $C$ is a set of $m$ *constraints*.

VARIABLES $X = \{x_1, \ldots, x_n\}$

Variables are containers modeling quantities that can refer to objects of the modeled problem. Variables can also be used solely for the purpose of modeling. These latter variables are automatically assigned if the decision variables are assigned. We often refer to variables corresponding to problem objects as *decision variables* while the variables used only for the model are simply referred to as *variables*.

DOMAINS $D = \{D(x_1), \ldots, D(x_n))\}$

To each variable $x_i$ is associated a domain $D(x_i)$ that is the set of possible values for $x_i$. The search for a solution for the CSP applies changes (reduction and restoration) to domains. Therefore, a distinction is made between the initial domain $D^{\text{init}}(x_i)$ of variable $x_i$ and its current domain $D(x_i)$. There are two ways to represent a domain:

either in extension where all the possible values are explicitly enumerated or with a range where only the lower and upper bounds of the domain are mentioned. The latter representation allows compact representation of domains but can only be used when there are no *holes* in the middle of the specified range.

CONSTRAINTS $C = \{c_1, \ldots, c_m\}$

A constraint $c_j$ restricts combination of values from a subset of variables from $X$. This subset of variables on which $c_j$ applies is referred to as the *scope* of the constraint $\text{scope}(c_j) \subseteq X$. The *arity* of a constraint $c_j$ is the cardinality of its scope: $\left|\text{scope}(c_j)\right|$.

The cross-product of domains of a subset of variables $Y \subseteq X$ are denoted $D(Y)$. An assignment of a set of variables $Y = \{y_1, \ldots, y_p\} \subseteq X$ is a set of pairs $\{(y_1, v_1), \ldots, (y_p, v_p)\}$ with $\{v_1, \ldots, v_p\} \in D^{\text{init}}(Y)$. An assignment is *valid* if the values are contained in the domains of the variables i.e., $\{v_1, \ldots, v_p\} \in D(Y)$. When an assignment of a set of variables $Y$ is such that it contains all the variables $X$ of the CSP (i.e., $Y = X$), we say that it is a *total* assignment. This implies that an assignment is total only if *all the variables* of the CSP are assigned to a value. An assignment of a set of variables $Y \subseteq X$ *satisfies* a constraint $c_j \in C$ if its restriction to variables from $\text{scope}(c_j)$ is such that $\text{scope}(c_j) \subseteq Y$ and is allowed by $c_j$. By extension, an assignment of variables satisfies a set of constraints $K \subseteq C$ if the assignment satisfies all the constraints in $K$. The purpose of a CSP resolution is to find a complete assignment such that all constraints are satisfied. In other words, CSP solving aims at finding a *total valid assignment*. In this thesis, the set of solutions from a CSP, $P = \text{CSP}(X, D, C)$, will be denoted $\text{sols}(P)$.

### 1.1.1 *CSP Example: The Graph Coloring Problem*

The graph coloring problem is a combinatorial problem that can be represented as a CSP. It aims at coloring the different regions delimited on a map such that two adjacent regions (regions sharing a border) are filled with different colors. When the number of available colors is much smaller than the number of regions to fill, this problem becomes hard to solve by hand. Figure 1.1 shows a map of the provinces of Belgium as an example. This problem can be represented as a graph-coloring problem [Myc55]. To do so, every delimited region becomes a vertex and there is an edge between two vertices if the two corresponding regions are adjacent. For example, the map from Figure 1.1 would

give the graph from Figure 1.2. A possible CSP representation for this problem could be the following one:

VARIABLES
There is one variable per delimited region (i.e., vertex in the corresponding graph) that has to be colored. Each variable represents the color that will fill the region. In the example from Figures 1.1 and 1.2, there are 10 variables, one per province:

$$X = \{\text{WF, EF, A, Lim, FB, WB, H, N, Liè, Lux}\}$$

DOMAINS
All variables have the same initial domain containing the set of colors available to fill the regions on the map. In the example from Figures 1.1 and 1.2, we consider that there are four available colors, leading to the following initial domains:

$$D^{\text{init}}(X) = \{\text{blue, orange, green, violet}\}$$

CONSTRAINTS
There is one constraint per pair of regions sharing a common frontier. In the example from Figures 1.1 and 1.2, the variable WF is in the scope of two constraints: WF $\neq$ EF and WF $\neq$ H. In the graph representation from Figure 1.2, each edge represent one of these constraints. This example contains 19 edges; hence, its CSP representation has a set of 19 constraints in total.

The example from Figure 1.1 contains 10 variables that can take 4 different values. A feasible solution to this example problem is shown in Figure 1.3. This very small example leads to a search space containing $10^4$ different potential assignments. Despite this large search space, a human should be able to find a solution in less than 30 seconds. It is because we tend to apply deductions and reasoning instead of enumerating all solutions. However, a human would take much longer to solve the same problem on all regions of countries from the EU. Fortunately, the Constraint Programming framework is able to solve large instances in a reasonably small amount of time using constraint propagation and search. Those two techniques are detailed in later sections.

## 1.2 CONSTRAINT OPTIMIZATION PROBLEM

A Constraint Optimization Problem (COP) is an extension of a CSP to which we add one or several optimization objectives. The aim of a COP

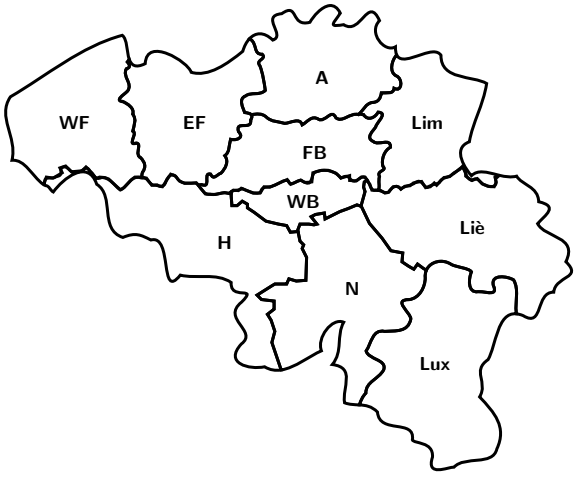



Figure 1.1: Example of map for the coloring problem. This is a map of Belgium divided into its ten provinces.

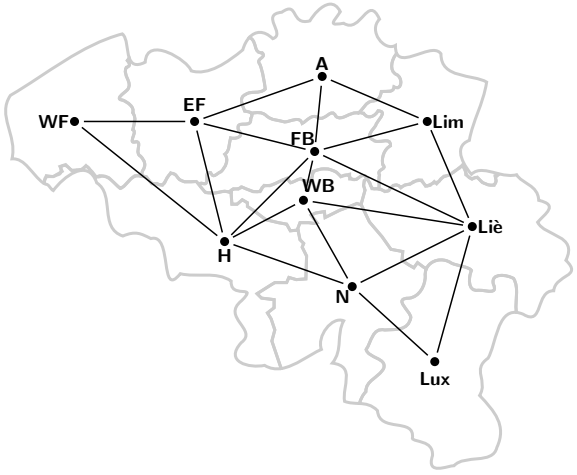



Figure 1.2: Graph representation of map from Figure 1.1 for the coloring problem. Each province is represented by a vertex and there is an edge between two provinces when they share a common frontier.

is to find an assignment of values to variables satisfying the constraints such that it optimizes one or several optimization functions. Formally, A COP is defined as a quadruplet $\mathrm{COP}(X, D, C, O)$ where $X$, $D$ and $C$ are respectively the set of variables, domains and constraints defined in Section 1.1 and $O$ is a set of objectives.

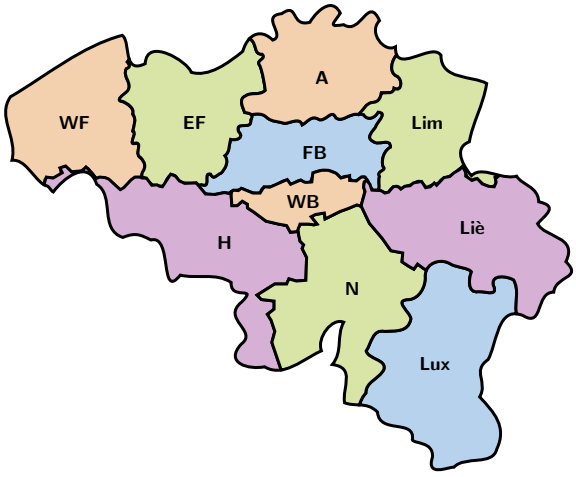



Figure 1.3: Possible solution of the map coloring problem from Figure 1.1.

OBJECTIVES $O = \{o_1, \ldots, o_l\}$

Objectives are functions whose output depends on a subset of variables from $X$. Each of these objectives has to be optimized, i.e., either minimized or maximized.

A clear distinction is made between techniques for single-objective and multi-objective optimization.

*Single-objective optimization* aims at finding *a single solution* optimizing a single objective function. When considering a single optimization objective, it is straightforward to compare two solutions and determine which one is more desirable. For example, when comparing two solutions $s_i, s_j \in \text{sols}(P)$ in a minimization context with a single objective function $f : \mathbb{R}^n \to \mathbb{R}$, $s_i$ is better than $s_j$ if $f(s_i) < f(s_j)$. Single objective optimization can aggregate several optimization criteria in a single objective function. Therefore, we will use the word *criteria* to refer to the different *goals* of the optimization. The term *objective* will be used to refer to the single objective function used in our model, possibly aggregating several optimization criteria. There are multiple possibilities to consider several criteria in single objective optimization. A first possibility is to aggregate all these criteria with a weighted sum. Another option is to order these criteria by decreasing preference; such objective criteria can thus be considered in *a lexicographical order*. When comparing two solutions, we first compare them according to the first (the most important) objective criterion. If the solutions cannot be discrimi-

nated on this criterion, we then compare them on the second objective, and so on.

*Multi-objective optimization* aims at finding *a set of solutions* that are tradeoffs between the multiple objective functions considered [Anj56]. Indeed, when comparing solutions in a multi-objective context, it is not always possible to determine a single best solution. Considering a set of objective functions $F \equiv \{f_1, \ldots, f_m\}$, a solution $s_i$ can be *more desirable* according to an objective $f_p$ than $s_j$ but oppositely $s_i$ can be *less desirable* than $s_j$ according to another objective $f_q$. The *Pareto dominance* [Par74] allows to evaluate if a solution is better than another solution with regards to several objective functions. In the context of minimization, the following definition of Pareto dominance is used. Considering two solutions $s_i, s_j \in \text{sols}(P)$, we say that solution $s_i$ dominates solution $s_j$ on objective functions $f_1, \ldots, f_m$, written $s_i \prec s_j$, if the two following conditions are satisfied:

$$s_i \prec s_j \equiv \begin{cases} \forall\, p \in \{1, \ldots, m\} : & f_p(s_i) \leq f_p(s_j) \\ \exists\, q \in \{1, \ldots, m\} : & f_q(s_i) < f_q(s_j) \end{cases}$$

Alternative dominance definitions exist, as those proposed in [KRK08, ZBT07, ALN99], but they are not detailed in this thesis. A solution $s_i$ is said to be Pareto optimal if it satisfies the following condition:

$$\nexists\, s_j \in \text{sols}(P) : \quad s_j \prec s_i$$

The Pareto optimal set is defined as the set of Pareto optimal solutions, i.e. the set of non-dominated solutions. Multi-objective optimization techniques aim at finding an approximation of this Pareto optimal set.

### 1.2.1 *COP Example: The Traveling Salesman Problem*

The Traveling Salesman Problem (TSP) [Lit+63] is a combinatorial problem that can be represented as a COP. Given a set of locations and their pair distances, the aim is to find the smallest route, in terms of distance, to visit all the locations in a single loop. As the number of locations increases, this problem becomes hard to solve by hand. Let us suppose as an example that Bernie Sanders, one of candidate for the USA presidency, decides to visit the ten most populated cities in the USA for his campaign. Once he has visited a city, he has to take a flight to pursue his tour to the next city. Figure 1.4 shows a map with the ten

most populated cities of the USA. As Bernie's campaign is based on ecology, he decides to make as few miles as possible in flight. Hence, he has to find the shortest circuit of direct flights between these cities. A possible COP representation for this problem could be the following one:

VARIABLES
There is one variable $s_i$ per location $i$ to visit. Each variable $s_i$ represents the successor location of $i$ i.e., $s_i$ is the location visited right after $i$. In the example from Figure 1.4, there are 10 variables, one per city on the map.

DOMAINS
All variables have an initial domain containing all the locations but themselves. In the example from Figure 1.4, there are ten cities and each initial domain thus contains nine cities.

CONSTRAINTS
There is a single constraint for this classic TSP. The `Circuit` constraint [Lau78] ensures that all locations are visited once (i.e., the succession of nodes forms a Hamiltonian cycle).

OBJECTIVES
This TSP is a single-objective optimization problem. The optimization objective is to minimize the total distance traveled. Formally, the objective function can be expressed as the sum of the distances traveled:

$$\min \sum_{i=1}^{n} \text{distance}(i, s_i)$$

where $\text{distance}(i, s_i)$ is the distance between location $i$ and location $s_i$.

The example from Figure 1.4 contains 10 variables that can each take 9 different values. A feasible solution to this example problem is shown in Figure 1.4. This very small example leads to a search space containing $10^9$ different potential assignments. If we take the cycle constraint into account, then the search space can be reduced to the number of permutations of 10 objects, leading to $10! \approx 3.63 \cdot 10^6$ potential assignments. For an arbitrary number of locations $n$, the problem has to consider $n!$ potential assignments. It is easily possible for a human to find a feasible solution such as the one from Figure 1.5. It is however very hard to find by hand the best solution and it is even harder to prove its optimality. Fortunately, as mentioned in Section 1.1, the CP

framework can solve large instances of TSP variants in a reasonably small amount of time [FLM02] with the help of constraint propagation and branching search.


Figure 1.4: Example of locations for the Traveling Salesman Problem. This is a map of the ten most populous cities in the USA.

## 1.3 CONSTRAINT PROPAGATORS

One of the main mechanisms of the Constraint Programming framework is called *constraint propagation* [RVW06, Apt03, Pie15]. Most of the time, the combinatorial problems tackled by CP have a very large search space. Constraint propagation is a mechanism allowing to remove parts of the search space that cannot contain any feasible solution. Every constraint stated in the model is associated to a propagation procedure that is in charge of finding unfeasible regions in the search space and removing them. The propagation procedure associated to each constraint is performed by one or several *propagators*. Propagators are algorithms taking as input a set of variables, the scope of the constraint it implements, and their current domains and returning a propagation outcome. Propagators attempt to remove inconsistent values from domains, i.e., values that provably will never be part of a feasible assignment with the remaining values in current domains. We refer to the removal of values from domains by a propagator as

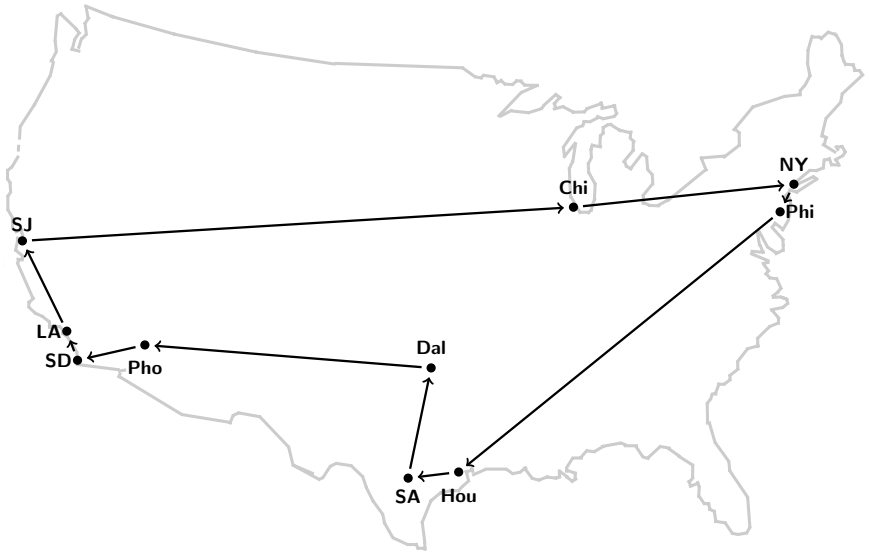



Figure 1.5: A possible solution for the TSP example from Figure 1.4. This solution, while having a high quality objective value, has not been proven to be the optimal solution of the example.

*pruning*. Whether or not a propagator has achieved pruning when it was called, it will return one of these three outcomes:

SUCCESS
: This outcome happens when a propagator will not be able to make any further deductions, whatever pruning is further achieved.

FAILURE
: A failure happens when the pruning of a propagator has led to an empty domain. If the domain of a variable is empty, it means that there is no possible assignment for this variable and thus, no solution.

SUSPEND
: If a propagator will potentially be able to achieve pruning when domains are further reduced and that no domain is empty, then it returns suspend as outcome.

Some constraints can be implemented by several propagators. The different propagators implementing a constraint can achieve different pruning i.e., they do not remove exactly the same values from domains of the variables. Even if some propagators are able to achieve more pruning than others, it often comes at the cost of a larger running time. During the resolution of a CSP or a COP, it can be useful to use several

propagators implementing the same constraint in order to achieve complementary pruning. It all boils down to a tradeoff between stronger pruning and larger amounts of time needed by propagators to run. As explained in [Smi05]: Maintaining a larger pruning, i.e., removing a larger proportion of inconsistent values, often takes more time; on the other hand, if more values can be removed from the domains of the variables, the search effort will be reduced and this will save time. Whether or not the time saved outweighs the time spent depends on the problem and the considered instance.

During the resolution of a CSP or a COP, the propagation procedure will take place until a fixed point is reached. A fixed point is reached when no constraint propagator can perform any more pruning on the current domains. To reach a fixed point and determine the call order of the different propagators, a *propagation queue* is used. The propagation queue allows to sort the propagators that have to be called. Usually, the propagation queue works as a FIFO queue i.e., the first propagator added to the propagation queue is the first to be popped out and run, while the last one added will be the last one to be popped out and run.

The propagators register to events occurring during a CP resolution. These events are various and the most common ones are:

- The bounds of a domain have changed, i.e., the minimum or maximum value from a domain has changed.
- A value has been removed from a domain.
- A variable has been bound, i.e., its domain contains a single element.

When an event to which a propagator is registered occurs, the propagator is added to the propagation queue. As the pruning performed by a given propagator can trigger several events, running a propagator can result in the addition of several propagators on the propagation queue. One of these propagators can also trigger events to which propagators are registered and so on. In fact, some propagators can trigger events causing themselves to be added again in the queue. The algorithm to reach the fixed point is a loop that iteratively pops out the first propagator from the propagation queue and runs it. The loop stops when the propagation queue is empty, meaning that the fixed point has been reached.

Algorithm 1.3.1 illustrates the fixed point algorithm. The main loop runs while the propagation queue is not empty. At line 2, the first

propagator is popped out from the propagation queue. The propagation of the propagator is performed in line 3 and we get back the list of events that has occurred. If the propagation has resulted in a failure, we stop the algorithm in line 5. Then, in line 9, we add to the propagation queue all the propagators that were registered to an event that has occurred during last propagation.

**Algorithm 1.3.1 :** Fixed Point Algorithm

```
1  while queue not empty do
2  |  propagator ← Pop(queue)
3  |  eventsTriggered ← Propagate(propagator)
4  |  if a failure has occurred then
5  |  |  Break();
6  |  end
7  |  foreach event in eventsTriggered do
8  |  |  foreach newPropagator registered to event do
9  |  |  |  Push(queue, newPropagator)
10 |  |  end
11 |  end
12 end
```

## 1.4 SEARCH

The Constraint Programming framework is often described with this simple equation:

$$\text{CP} = \text{Model} + \text{Search}$$

This expresses that a CP model is decoupled from the search strategy used to solve it. A given model can be solved using different search strategies; similarly, a given search strategy can be used to solve different models. The model in our equation is either a CSP or a COP, described respectively in Sections 1.1 and 1.2. The search in our equation is defined by a branching search strategy.

Branching search strategies used in CP are *constructive* methods. They can be either *complete* or *incomplete*, meaning they respectively explore the whole search space or only a part of it. A branching divides a problem $P = \text{CSP}(X, D, C)$ into a set of smaller subproblems $\{P_1 = \text{CSP}_1(X, D, C_1), \ldots, P_r = \text{CSP}_r(X, D, C_r)\}$. A subproblem

$P_i = \text{CSP}_i(X, D, C_i)$ is obtained by the addition of a set of new constraints to the problem $P = \text{CSP}(X, D, C)$. Hence, a subproblem $P_i$ is obtained by constraining even more the original problem $P$. This implies that all the possible solutions of the subproblem $P_i$ are also solutions of the original problem $P$: $\text{sols}(P_i) \subseteq \text{sols}(P)$. The original problem is partitioned such that the solution sets of the subproblems are disjoint:

$$\forall\, i, j \in [1, \ldots, r] : \text{sols}(P_i) \cap \text{sols}(P_j) = \varnothing$$

When the branching strategy is complete, the union of the subproblems cover all the search space from the original problem:

$$\bigcup_{i=1}^{r} \text{sols}(P_i) = \text{sols}(P)$$

On the other hand, if the branching strategy is incomplete, there is no guarantee that this last equation holds.

A recursive application of the branching strategy builds a *search tree*. Starting from a root node problem $P$ (depth 0 in the tree), children nodes are obtained with a recursive division into new subproblems (depth 1). These subproblems are recursively divided into smaller subproblems (depth 2). Nodes at each depth are then recursively divided into children nodes at an incremented depth. The division of a problem into subproblems is called *branching*.

When a branching strategy is complete, is corresponds to a complete enumeration of all the potential assignments of values to variables of the original (root) problem. To reduce the time taken by the search, it is important to remove pieces of the search space that will never contain any feasible solution. When a new node is created, constraint propagation is applied on its corresponding CSP until a fixed point is reached. This constraint propagation reduces the search space, potentially removing some subproblems that would have been explored otherwise.

The recursive division of a node into subproblem children nodes stops when either one of these two situations occurs:

1. Constraint propagation resulted in a failure. This occurs when a domain becomes empty. This means that the CSP in this node contains no feasible solution. As such, this node does not have to be explored.

2. A solution has been found. This means that after constraint propagation all domains contain a single value. The search can thus stop as a feasible solution has been reached or continue if all solutions are queried.

When solving a COP, the search should not stop when a solution has been reached. Instead, a search strategy called *branch and bound* is used. The branch and bound strategy maintains bounds on the value of the objective function. Additionally to classic branching search, it constrains the objective value to lie within these bounds. As the objective is to find the best solution according to the objective function, the bounds will be tightened during the search. Whenever a new solution is found, the bounds are tightened such that no solution can have a worse objective value than the new discovered solution. Hence, it can happen that nodes leading to feasible solutions (in the original problem) are not expanded because they lead to solutions with a worse objective value.

While the branching strategy determines the *shape* of the search tree, the *search strategy* defines in what order the nodes in the search tree are explored. The exploration of the tree is driven by an ordered set of *open nodes* called the *frontier*. An open node is a node representing a subproblem $P$ that has not been yet be divided into subproblems. On the other hand, a *visited* node is a node that has already been divided into subproblems. The iterative traversal of the search tree removes the first open node from the frontier, divides it into subproblems, and adds the corresponding children nodes to the frontier. When the frontier is empty, the whole search tree has been traversed. The ordering of the open nodes in the frontier is defined by the branching strategy. A simple *Depth-First Search* strategy orders nodes in the frontier such that the last node added is the first to be visited. A more elaborated *Heuristic Search* strategy associates a *heuristic* value to each node in the frontier. The frontier orders the nodes by increasing heuristic value.

### 1.4.1 *Search Example*

To illustrate the search mechanisms in CP, we will consider the following small CSP example:

VARIABLES

There are three variables:

$$X = \{x_1, x_2, x_3\}$$

DOMAINS

All three variables have the same initial domains containing only 3 values:

$$D^{\text{init}}(X) = \{1, 2, 3\}$$

CONSTRAINTS

There are only three constraints:

$$x_1 > x_2$$

$$x_1 > x_3$$

$$x_2 \neq x_3$$

Figure 1.6 shows an example of search three for this small CSP example. Here are the different steps followed in the tree traversal:

INITIAL PROPAGATION

Initial propagation at the root node removes the value 1 from $D(x_1)$ and the value 3 from both $D(x_2)$ and $D(x_3)$ since these values cannot satisfy the constraints $x_1 > x_2$ and $x_1 > x_3$.

BRANCHING $x_1 = 2$

The propagation of constraints $x_1 > x_2$ and $x_1 > x_3$ removes the value 2 from both $D(x_2)$ and $D(x_3)$; the propagation of constraint $x_2 \neq x_3$ then leads to a failure since both $D(x_2)$ and $D(x_3)$ now contain the same single value: 1.

BRANCHING $x_1 \neq 2$

This leaves a single value in the domain of $x_1$: $D(x_1) = \{3\}$. The propagation of the constraints does not remove any unfeasible value.

BRANCHING $x_2 = 1$

The propagation of constraint $x_2 \neq x_3$ removes 1 from $D(x_3)$. All variables are assigned and we have found a solution: $D(x_1) = 3, D(x_2) = 1, D(x_3) = 2$.

BRANCHING $x_2 \neq 1$

This leaves a single value in the domain of $x_1$: $D(x_2) = \{2\}$. The propagation of constraint $x_2 \neq x_3$ removes 2 from $D(x_3)$. All variables are assigned and we have found a solution: $D(x_1) = 3, D(x_2) = 2, D(x_3) = 1$.

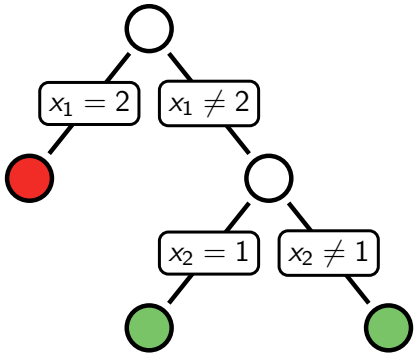



Figure 1.6: A possible tree traversal for the small CSP example. Red nodes are drawn when constraint propagation led to a failure. Green nodes are drawn when a solution has been reached.

## 1.5 LARGE NEIGHBORHOOD SEARCH

When considering a COP with a very large search space, it is often too hard to find an optimal solution. In such cases, it is interesting to lose the completeness of an approach to allow faster discovery of hopefully higher quality solutions. As such approaches are not complete, even if we find a high quality solution, we cannot prove that it is optimal. However, in many applications, optimality is not required as long as the best solution obtained has a sufficient objective quality.

Large Neighborhood Search (LNS) [Sha98] is a metaheuristic that can be applied to many frameworks, including CP. It proposes to lose the completeness of an approach to orient the search in promising areas of the search space. LNS has proven to be very efficient to solve some hard COP, several of them implemented with CP + LNS. The main idea of LNS is to take advantage of the best solution so far to quickly find new solutions of hopefully better quality. LNS iteratively applies two phases: *relaxation* and *reconstruction*:

### RELAXATION

Starting from the best solution found so far, the relaxation proposes to *relax* parts of the solution, i.e., reset parts of the solution to their initial states from the original COP. The parts that are relaxed are often driven by heuristics, including randomness. This ensures that successive relaxations starting from the same solution will not lead to the same CSP. The CSP obtained after relaxation should hopefully be much smaller than the original CSP. Furthermore, it contains parts of the best solution and as such looks promising.

RECONSTRUCTION
Reconstruction attempts to solve the CSP obtained after a relaxation phase. As some part of the CSP are already fixed, the search space of this CSP is smaller and the time needed to reach failures or new solutions will hopefully also be smaller.

If the reconstruction finds a new solution of higher quality, then a new iteration relaxation-reconstruction with this new solution begins. On the other hand, if the reconstructions fails to find a better solution in a fixed amount of time, then a new iteration begins from the former solution with a different relaxation.

The choice of the relaxation to perform depends on many factors, such as the type of problem, the number of iterations already performed, etc. Several works have proposed ways to select the relaxation to perform at each iteration. Mairy et al. [MDH11] propose a reinforcement learning method to select the subset of variables to relax at each LNS iteration. Laborie and Godard [LG07] propose another method allowing, given a set of possible relaxations, to select at each LNS iteration what relaxation to apply. In [PP09, Cle+10], a *soft-cumulative* constraint is presented that allows to associate a cost to the usage a cumulative resource. It defines an *ideal capacity*, below the max capacity bounding the maximal usage of the resource, representing the desired usage of a cumulative resource. The goal is to minimize a function of the differences at each time step between the usage of a resource and the ideal capacity.

### 1.5.1 *LNS Example: The Traveling Salesman Problem*

The Traveling Salesman Problem described in Section 1.2 can be solved with the help of Large Neighborhood Search. We present here an example of TSP resolution using CP + LNS. The CP model used is the same as the one described in Section 1.2. We consider the resolution of the example with the candidate to presidency campaign tour from Figure 1.4.

As an example, let us consider that the CP search eventually finds the solution from Figure 1.5. Starting from this solution, LNS will perform iterations of relaxation and reconstruction. A possible relaxation for our example could be to randomly relax 30% of the variables $s_i$ i.e., 30% of the variables have their domain set back to their initial domains,

while the other variables remain assigned to values from the last solution found. An example of possible relaxation is shown in Figure 1.7.

From this partial assignment, the reconstruction using classic CP resolution will eventually find a new solution shown in Figure 1.8. If this solution has a better objective value than the previous one, then new LNS relaxations will start from this new solution. Otherwise, new iterations will begin starting from the former solution from Figure 1.5 until a better solution is found.

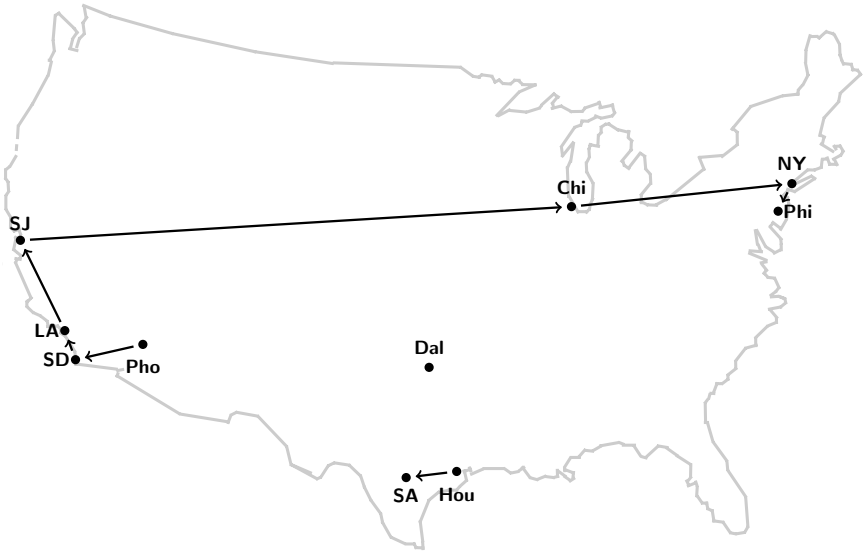



Figure 1.7: A possible relaxation of 30% of the variables starting from solution from Figure 1.5.

## 1.6 OTHER COMBINATORIAL OPTIMIZATION TECHNIQUES

There exist a wide range of combinatorial optimization resolution techniques [Coo+11, PS82] other than CP. Every technique has its own advantages and its own strategy to solve problems. Some techniques are well known to be efficient on well known classes of problem. This section aims at mentioning the most known alternatives to CP and their characteristics and differences. We present here two additional optimization techniques: Integer Linear Programming and Local Search.

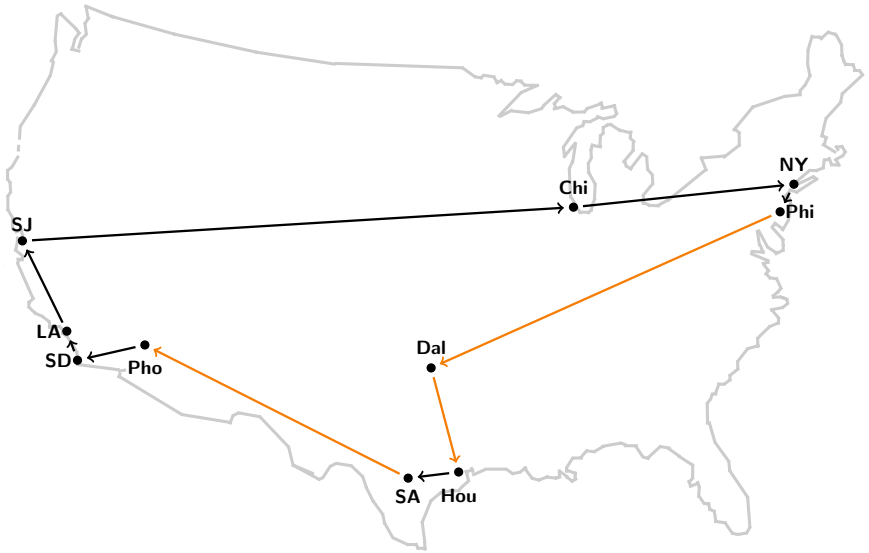



Figure 1.8: A possible solution built from the partial solution from Figure 1.7. The new assignments are shown in orange.

### 1.6.1 *Integer Linear Programming*

In [WN14], Integer Linear Programming (ILP) refers to the set of techniques aiming at resolving problems that can be expressed as integer programs. Integer programs are models for problems where variables can only take integer values and constraints are linear (in)equations. A *feasible* solution is a complete assignment of decision variables to integer values such that the constraints are respected. [WN14] presents several integer linear programming techniques: the branch and bound method [LW66], the cutting planes method [Kel60], etc. Apart from some specific variants, Integer Linear Programming resolution techniques are complete (i.e., they explore the whole search space and provide a proof of optimality). Furthermore, ILP is an exact method since it does not exclude any feasible solution.

### 1.6.2 *Local Search*

Local Search (LS) [RN95] is a method used to solve hard combinatorial problems. It is a perturbative method: starting from an existing solution, it perturbates it to reach another solution hopefully of higher quality. The next solution is chosen from the *neighborhood* of the cur-

rent solution. The choice of the solution selected from the neighborhood of a solution is driven by heuristics. Local Search techniques can be trapped inside local optima. In order to escape local optima, restart strategies and metaheuristics can be used. Some well known LS metaheuristics are simulated annealing [Kir83], tabu search [Glo86], ant colony optimization [Sol10] and genetic algorithms [RN95]. LS is incomplete and as such does not provide any proof of optimality. LS is inexact since it will not always browse *all* feasible solutions.

### 1.6.3 *Comparison of Resolution Techniques*

We propose here a summary of the tree resolution techniques we have introduced here. We propose to differentiate these techniques on two main aspects. First, we distinguish *constructive* from *perturbative* methods. A method is constructive if it builds a solution piece by piece through the exploration of a search tree. On the other hand, perturbative methods iteratively modify an existing solution to find a new one. Second, we distinguish *complete* and *incomplete* methods. A method is complete if it explore the whole feasible search space, while it is incomplete if it does not. A summary of the techniques described in this chapter is shown in Table 1.1.

| | Complete | Incomplete |
|---|---|---|
| Constructive | CP<br>ILP | |
| Perturbative | | LS |

Table 1.1: Comparison of combinatorial optimization resolution techniques.

# 2

# SCHEDULING

*Next Saturday night, we're sending you back to the future!*

—Dr. Emett Brown, *Back to the Future*

*We must use time creatively.*

—Martin Luther King, Jr.

*Coming back to where you started is not the same as never leaving.*

—Terry Pratchett, *A Hat Full of Sky*

*Most people spend more time and energy going around problems than in trying to solve them.*

—Henry Ford

*Lost time is never found again.*

—Benjamin Franklin

Scheduling [BLN01] is a generic term covering a wide family of optimization problems. Many real-world problems can be described as scheduling problems. However, scheduling problems tend to be hard to solve and are computationally expensive. Hence, many researches address scheduling and it is still widely studied nowadays. These researches have led to a broad variety of resolution techniques to tackle scheduling problems. Scheduling problems aim at placing a series of events in *time* such that these events satisfy a set of constraints. This global definition will be clarified in Section 2.1 where we give a formal definition of scheduling problems. In Section 2.2 we describe the different categories of scheduling problems. Finally, Section 2.3 introduces a small scheduling problem example.

## 2.1 SCHEDULING PROBLEMS

A scheduling problem can be described as defining *when* a set of activities are executed such that a set of constraints are respected. Solutions of scheduling problems are referred to as *schedules*, that is an exact definition of when activities start and end such that constraints are respected. Sometimes, a given scheduling problem can have multiple solutions. Most of the time, some schedules are more desirable than others according to one or several optimization criteria. Hence, most scheduling problems are optimization problems. Even if time is continuous, it is discretized in most scheduling applications. In this thesis, we consider that time is discretized into supposedly indivisible small steps. The discretization steps may vary from seconds to days. This implies that the attributes related to time will be defined with integers. Therefore, the events that have to take place in time will always start and end at integer times. Scheduling applications link these integer times with real time units (seconds, minutes, days, months, years, ...). Furthermore, we shift the schedules to a time window starting from 0 and ending at the *horizon*. No event can start before 0 nor any event can end after the horizon. This time window can be shifted back to the original bounds of the schedule (e.g., 0 corresponds to 9.40 a.m. on $10^{th}$ January 2016 and horizon corresponds to 6.50 p.m. on $11^{th}$ January 2016).

Most scheduling problems can be entirely defined by a set of components. Even though several different components are used in the literature, we propose a definition of scheduling problems with three

main components: activities, constraints and objectives. We will review these three main concepts in more details in this section.

### 2.1.1 *Activities*

The events that have to be placed in time are referred to as *activities* (sometimes they are also called *tasks*). An activity is an "event" that can be executed at some point in time. Activities will be written in this thesis with a capital letter (most of the time "A") coupled with subscript indices, e.g., $A_1, A_j, A_{k,l}$. An activity $A_i$ encapsulates three CP variables. The first variable $d_i$ represents the duration of the activity. Then, the start variable $s_i$ represents the moment at which the activity begins its execution. Similarly, the end variable $e_i$ represents the time at which the activity is completed. The initial domain of these variables is described with a set of attributes:

DURATION $\text{dur}_i$
The duration of an activity corresponds to the amount of time during which it has to be executed. It can either be a fixed integer $\text{dur}_i$ or it can be bounded by a minimal and maximal integer value, respectively $\min(\text{dur}_i)$ and $\max(\text{dur}_i)$.

RELEASE DATE $\text{rel}_i$
The release date of an activity is a point in time before which it cannot be executed. It is an integer fixed value written $\text{rel}_i$.

DEADLINE $\text{dea}_i$
The deadline of an activity is a point in time after which it cannot be executed. It is an integer fixed value written $\text{dea}_i$.

With these attributes, the variables associated to an activity $A_i$ have initial domains defined by the following ranges:

$$\begin{aligned} D^{\text{init}}(d_i) &= [\min(\text{dur}_i), \max(\text{dur}_i)] \\ D^{\text{init}}(s_i) &= [\text{rel}_i, \text{dea}_i - \min(\text{dur}_i)] \\ D^{\text{init}}(e_i) &= [\text{rel}_i + \min(\text{dur}_i), \text{dea}_i] \end{aligned}$$

Furthermore, these three variables are linked together by the following constraint:

$$s_i + d_i = e_i$$

In addition to the attributes defined earlier, an activity $A_i$ can also be described by a set of features:

PREEMPTIVE
An activity is *preemptive* if its execution can be interrupted and then restarted later. On the opposite, a *non-preemptive* activity has to be processed at one go and cannot be interrupted. A preemptive activity has to be executed completely even if it is interrupted. This means that a preemptive activity that has been interrupted restarts its execution at the point at which it was interrupted.

OPTIONAL
An activity is *optional* if it is not mandatory to be executed. On the opposite, an activity is *mandatory* if it must be executed. If an optional activity is executed, it has to be completely executed during the schedule; there is no such things as partially executed activities.

In this thesis, we will only consider non-preemptive mandatory activities. This means that every activity described in our models has to be executed and cannot be interrupted. Additional decision variables are often used to express whether an activity is optional or not and whether it is preemptive or not. We do not introduce them as we will not consider preemptive nor optional activities in this thesis.

A graphical representation of an activity and its attributes is shown in Figure 2.1. Activities are often represented in a *Gantt chart* [Wil03]. Gantt charts are unidimensional graphs in which the horizontal axis represent time and activities are represented as 2D rectangles such that their length corresponds to their duration. Furthermore, the rectangle starts along the axis when the activity begins to be executed and ends when the activity has been completed. Their vertical placement is often made such that structured group of activities are on the same line and such that rectangles do not overlap each other.

The variables $d_i$, $s_i$ and $e_i$ are bounded by four quantities commonly used to describe the potential placement in time of the corresponding activity $A_i$ in a current node:

EARLIEST STARTING TIME $\text{est}_i$
The earliest starting time of an activity corresponds to the first point in time at which an activity can begin its execution. Formally, $\text{est}_i$ represents the minimal value of the current domain of variable $s_i$:

$$\min(D(s_i)) = \text{est}_i$$

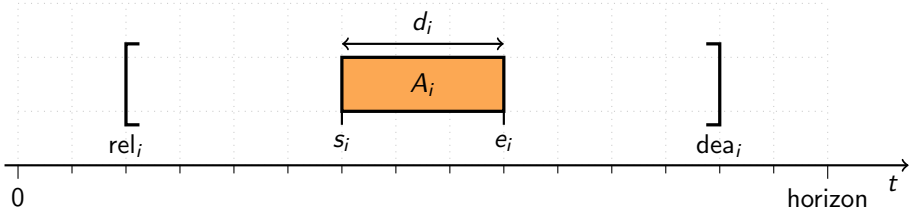



Figure 2.1: Graphical representation of an activity $A_i$ with its attributes and decision variables. It can be scheduled between the opening and closing brackets representing respectively the release date $\text{rel}_i$ and deadline $\text{dea}_i$ of the activity.

LATEST STARTING TIME $\text{lst}_i$

The latest starting time of an activity corresponds to the last point in time at which an activity can begin its execution. Formally, $\text{lst}_i$ represents the maximal value of the current domain of variable $s_i$:

$$\max(D(s_i)) = \text{lst}_i$$

EARLIEST COMPLETION TIME $\text{ect}_i$

The earliest completion time of an activity corresponds to the first point in time at which an activity can end its execution. Formally, $\text{ect}_i$ represents the minimal value of the current domain of variable $e_i$:

$$\min(D(e_i)) = \text{ect}_i$$

LATEST COMPLETION TIME $\text{lct}_i$

The latest completion time of an activity corresponds to the last point in time at which an activity can end its execution. Formally, $\text{lct}_i$ represents the maximal value of the current domain of variable $e_i$:

$$\max(D(e_i)) = \text{lct}_i$$

These quantities describing the potential placement in time of an activity $A_i$ allow to perform reasoning with other activities. This reasoning helps to reduce the search space when solving a scheduling problem. Some techniques used to reduce the potential placement in time of activities will be detailed in later chapters. As such, when a propagation mechanism is used to update these four values, the corresponding updates are also applied to the related variables. These four quantities are represented graphically in Figure 2.2.

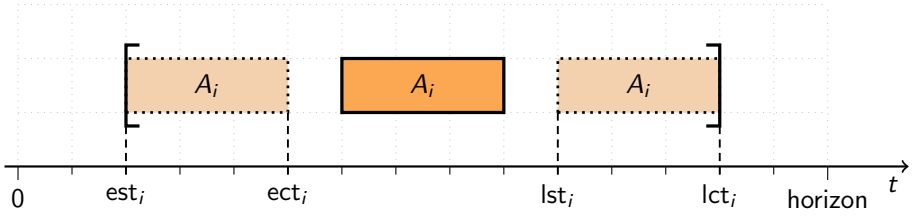



Figure 2.2: Graphical representation of an activity $A_i$ and the four quantities deduced from its decision variables. The quantities $est_i$ and $ect_i$ are obtained when placing $A_i$ at the first possible point in time. Similarly, $lst_i$ and $lct_i$ are obtained when placing $A_i$ at the last possible point in time.

### 2.1.2 *Constraints*

Scheduling problems are subject to a wide range of constraints that can be separated in three different categories. The first category contains the time related constraints that constrain the time at which activities can be scheduled related to each other. The second category contains the resource constraints in which activities need to use a resource in order to be executed. Finally, the third category contains all the additional constraints that are more general and not restricted to scheduling problems. We will not describe constraints from this latter category in this thesis. We describe the two first categories of constraints, time-related and resource constraints, in this section.

#### *Time-Related Constraints*

The first important time-related constraint is called the *precedence* constraint. An activity $A_i$ precedes $A_j$, written $A_i \ll A_j$, if $A_i$ has to be completely executed before $A_j$ starts its own execution. This implies that $A_j$ cannot start before $A_i$ is finished. Formally, $A_i$ precedes $A_j$ can be expressed as follows:

$$e_i \leq s_j$$

For example, let us consider the baking of a cake as a scheduling problem. This problem has only two activities: mixing the ingredients together, and put the mix in the oven for 30 minutes. This problem contains a precedence constraint: the activity mixing the ingredients precedes the activity where the cake is put in the oven for 30 minutes.

Another widely used time-related constraint is the *transition times* constraint. This constraint states that two activities $A_i$ and $A_j$ cannot

be executed simultaneously and a minimal amount of time must take place between their respective executions. The amount of time that must occur between the two activities is not symmetrical and depends on which activity is executed before the other. If $A_i$ is executed before $A_j$, then there must be at least a transition time $\mathrm{tt}_{i,j}$ between their respective executions. On the other hand, if $A_j$ is executed before $A_i$, then there must be at least a transition time $\mathrm{tt}_{j,i}$ between their respective executions. Formally, a transition time constraint between two activities $A_i$ and $A_j$ is expressed as follows:

$$e_i + \mathrm{tt}_{i,j} \leq s_j \quad \vee \quad e_j + \mathrm{tt}_{j,i} \leq s_i$$

For example, let us consider the baking of two different cakes as a scheduling problem. The first cake must be put in the oven at a temperature of 200 °C for 20 minutes: this is activity $A_i$. Similarly, the second cake must be put in the oven at a temperature of 175 °C for 40 minutes. The oven needs 5 minutes to decrease its temperature from 200 °C to 175 °C. Similarly, the oven needs 10 minutes to increase its temperature from 175 °C to 200 °C. These two different amounts of time needed to adjust the oven temperature can be modeled with transition times.

*Resource Constraints*

Many scheduling problems involve the use of resources. A resource is an "object" that is needed by an activity during its execution. There exist several different resources and they involve different constraints. Most of the time, resources are shared by a set of activities $\Omega$. As such, the constraint implied by a resource involves all the activities using it. Therefore, the constraint implied by the resource contains all the variables from the activities using it in its scope. In this section, we describe the four main types of resources that can be used to model scheduling problems along with the constraints they imply:

UNARY RESOURCE

A unary resource [Vil04a] can be used by a single activity at any point of time. It is sometimes referred to as *disjunctive resource* or *machine*. A unary resource is used by an activity during its whole execution, forbidding any other activity to use it during this period. Two activities using a unary resource cannot overlap in time and this induces a disjunction of precedence constraints between those. Formally, a group

of activities $\Omega$ using a unary resource is subject to the following constraints:

$$\forall_{i \neq j} \; A_i, A_j \in \Omega : \; e_i \leq s_j \; \vee \; e_j \leq s_i$$

An example of unary resource could be a sawing machine used to split large planks into smaller planks. The machine can only cut a single plank at a time. Hence, if cutting a large plank is modeled by an activity, then the sawing machine is modeled with a unary resource.

#### CUMULATIVE RESOURCE

A cumulative resource [Bel+96, BC02] can be used by several activities simultaneously up to a determined capacity. Each activity uses a determined amount of the resource abusively called *height*. Cumulative resources can be separated in two distinct categories: renewable resources and non-renewable resources.

#### RENEWABLE RESOURCES

A renewable resource, sometimes referred to as *discrete resource*, allows multiple activities to use it simultaneously, as long as the resource capacity is not exceeded. When an activity begins, it acquires a given amount of the resource is acquired; this amount is returned at the end of its execution. Hence, the available quantity of a renewable resource is reduced by an activity only during its execution. Concretely, a non-renewable resource imposes the following constraint: at any point of time, the sum of the heights of activities currently executed has to be lower or equal to the resource capacity. Formally, a group of activities $\Omega$ in which each activity $A_i$ needs a height $h_i$ of a cumulative resource of capacity $C$ is subject to the following constraint:

$$\forall \, t \in [0, \text{horizon}] : \sum_{\substack{A_i \in \Omega \\ t \in [s_i, e_i[}} h_i < C$$

Following the terminology used in the ILOG solver [Lab09], the cumul function representing the use of a renewable resource is the *pulse* function. The pulse function $\text{pulse}(A, h)$ takes the value $h$ (representing the height i.e., the amount of resource used by activity $A$) during the execution time of activity $A$ and is null otherwise. A graphical representation of the pulse function is shown in Figure 2.3. An example of renewable resource could be a team of workers. Several actions, each demanding a different number of persons, have to

be performed. The total required number of persons by the different actions executed at any point in time can never exceed the number of persons from the team of workers. Hence, the team of worker can be modeled by a renewable resource.

NON-RENEWABLE RESOURCE

A non-renewable resource, sometimes referred to as *reservoir resource* is a consumable resource whose *level* evolves through time as activities produce or consume amounts of it. When an consumer activity begins, it consumes a given amount of the resource that is never returned. When an producer activity ends, it provides a given amount of the resource that is never taken back. Consumer activities lower the level of the resource while producer activities increase it. Each activity associated with the resource is either consumer or producer but cannot be both. As the resource and its level is discrete, activity consume or produce only integer amounts of it. The convention is that consumer activities decrease the resource level of their height at the moment they start their execution. On the other hand, producer activities increase the resource level at the moment they end their execution. At every point of time $t$, the level of the reservoir resource $l_t$ is bounded by a minimal and a maximal level, respectively $L_{\min}$ and $L_{\max}$. Formally, the resource constraint can be expressed as follows:

$$\forall\, t \in [0, \text{horizon}] :\ L_{\min} \leq l_t \leq L_{\max}$$

Following the terminology used in the ILOG solver [Lab09], the cumul function representing the use of a non-renewable resource is the *step* function. The step function $\text{step}(A, h)$ is null until activity $A$ starts where it takes the value $h$ (representing the height i.e., the amount of resource used by activity $A$) until the horizon. A graphical representation of the step function is shown in Figure 2.4. An example of reservoir resource could be a tank of fuel. Some activities would pour fuel in the tank, increasing its level while other activities would use fuel, decreasing the level of the tank. The minimal level of the resource corresponds to the empty tank and the maximal level of the resource is the volume that the tank can contain.

Graphically, one can represent the usage of a cumulative resource (either renewable or not) with a cumulative profile. The height of rectangles representing activities using a cumulative resource represents the amount of resource it consumes (or produces in the case of a non-renewable resource). In a cumulative profile, the horizontal axis

represents the time and the vertical axis indicates the amount of resource used. An example of schedule and the associated cumulative profile of a renewable resource are represented in Figure 2.5. Similarly, an example of schedule and the associated cumulative profile of a non-renewable resource is represented in Figure 2.6.

ALTERNATIVE RESOURCE

An alternative resource [FLN00, AB93] defines a set of either unary or cumulative resources. An activity using an alternative resource must be assigned to one of the possible resources defined by the alternative resource. The resource to which an activity is assigned must have a sufficient amount to satisfy the demand of the activity at the time at which it will be executed.

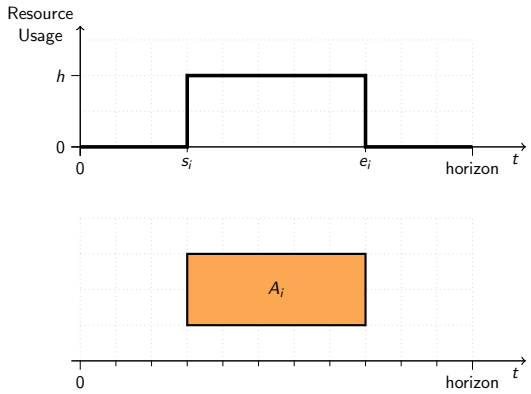



Figure 2.3: Illustration of the pulse$(A_i, h)$ cumul function representing the consumption of a renewable resource by an activity $A_i$.

### 2.1.3 *Objectives*

There is a large panel of possible objective functions for scheduling problems. In this section, we will focus on classic scheduling objectives. These classic scheduling objectives can be defined as combinations of the end decision variables described earlier. Some objective functions are defined according to due-dates associated to a subset of activities. A due-date $\text{due}_i$ is a preferential time at which an activity $A_i$ should end. Note that a due-date is a *preference* i.e., it does not imply a constraint. On the other hand, as specified earlier, a *deadline* imposes a constraint. The main components of objective functions associated to an activity $A_i$ are the following ones:

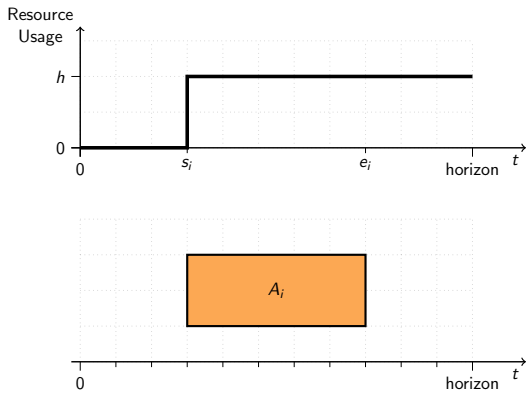



Figure 2.4: Illustration of the step$(A_i, h)$ cumul function representing the consumption of a non-renewable resource by an activity $A_i$.

#### ENDING TIME

As described before, it is the time at which an activity ends.

#### LATENESS

The lateness of an activity is defined as the amount of time between the due-date of an activity and its actual ending time. The lateness of an activity can be negative if the activity ends before its due-date. Formally, the lateness of an activity is defined as follows:

$$\text{late}_i = e_i - \text{due}_i$$

#### TARDINESS

Similarly to lateness, the tardiness of an activity is defined as the amount of time between the due-date of an activity and its actual ending time. However, unlike lateness, tardiness only takes into account activities ending *after* their respective due-dates. If an activity ends its execution before or at its due-date, its tardiness is null. Formally, the tardiness of an activity is defined as follows:

$$\text{tard}_i = \max(0, e_i - \text{due}_i)$$

Sometimes, the actual amount of time between the due-date and the ending of an activity is not important. In such situations, a fixed penalty is associated to an activity ending after its due date. Such objective is called *reified tardiness*, reif$(\text{tard}_i)$, and is formally expressed as follows:

$$\text{reif}(\text{tard}_i) = \begin{cases} 1 & \text{if } e_i > \text{due}_i \\ 0 & \text{otherwise} \end{cases}$$

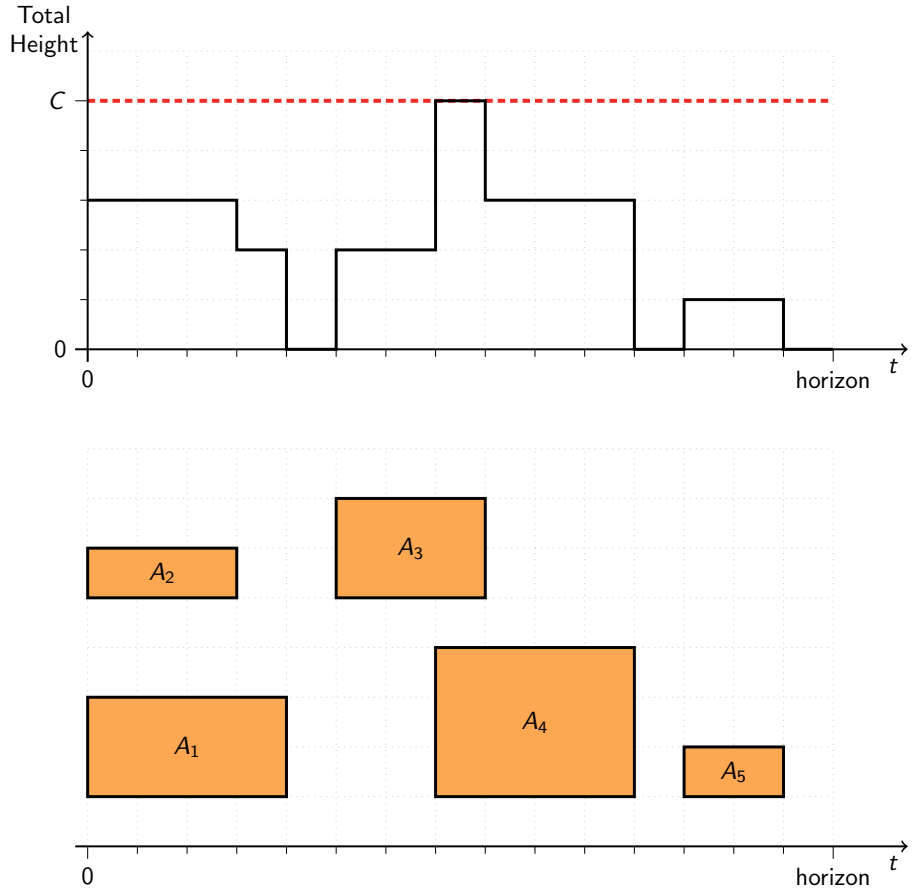



Figure 2.5: Schedule of activities using the same renewable resource and its associated cumulative profile. The line in the cumulative profile corresponds to the sum of the heights of activities executed at any point in time.

### EARLINESS

Earliness is similar to tardiness; the earliness of an activity is defined as the amount of time between the actual ending time of an activity and its due-date. However, on the opposite of tardiness, earliness only takes into account activities ending *before* their respective due-dates. If an activity ends its execution at or after its due-date, its earliness is null. Formally, the earliness of an activity is defined as follows:

$$\text{earl}_i = \max(0, \text{due}_i - e_i)$$

Similarly to tardiness, there is a reified version of earliness in which a fixed penalty is associated to an activity ending before its due date.

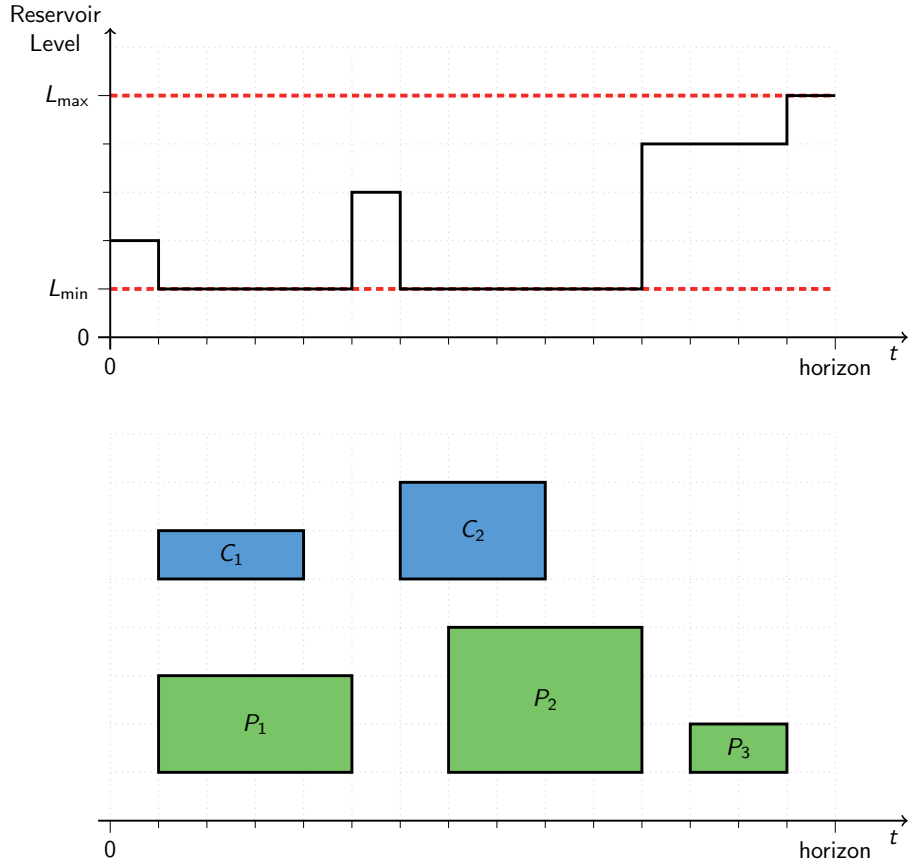



Figure 2.6: Schedule of activities using the same non-renewable resource and its associated cumulative profile. The activities written with a capital 'P' are producers and activities written with a capital 'C' are consumers. The line representing the resource level goes up at the end of producer activities and goes down at the beginning of consumer activities.

Such objective is called *reified earliness*, $\text{reif}(\text{earl}_i)$, and is formally expressed as follows:

$$\text{reif}(\text{earl}_i) = \begin{cases} 1 & \text{if } e_i < \text{due}_i \\ 0 & \text{otherwise} \end{cases}$$

The most widely known objective functions for scheduling problems can be defined with these quantities. First, the *makespan* represents the width of the time window where the schedule lies. The objective is to

find the minimum makespan for a scheduling problem. Formally, it is defined as the maximal ending time of the set of all activities in the schedule:

$$\text{minimize} \quad \text{makespan} = \max_{A_i \in \mathcal{A}} e_i$$

where $\mathcal{A}$ is the set of all the activities in the considered problem.

Other well known objective functions are weighted sums of the earliness and tardiness (or their reified versions). This models the preference of placement of some activities in possible schedules. Formally, the objective is to minimize a weighted sum of earliness and tardiness of activities:

$$\text{minimize} \quad \sum_{A_i \in \mathcal{A}} we_i \cdot \text{earl}_i + wt_i \cdot \text{tard}_i$$

where $we_i$ and $wt_i$ are weights attached respectively to the earliness and tardiness of an activity $A_i$.

Finally, some problems associate a cost to the usage of a resource. In some problems, the resource constraints are considered *soft constraints*. This means that their capacity can be exceeded but the objective is to limit the number of times and/or the amount by which the resource capacities are exceeded. We

## 2.2 CATEGORIES OF SCHEDULING PROBLEM

In Section 2.1, we have presented the three main components defining a scheduling problem: activities, constraints and objectives. Scheduling problems with similarities in those components can be grouped into problem categories. Graham has proposed a classification of scheduling problem in [Gra+79]. A web tool called *Scheduling Zoo* [DK13] proposes to list scientific publications tackling a scheduling problem according to its characteristics. This tool uses the problem classification defined by Graham. In this section, we propose to enumerate some widely used scheduling problem categories using an approach similar to [Mon10, Dej12].

### 2.2.1 *Shop Problems*

Shop Problems are problems in which activities are grouped in jobs. In the context of scheduling, a *job* is defined as a group of activities sharing characteristics or to which some structure is imposed. It most of the time considers $n$ jobs containing $m$ activities with demands on

$m$ different resources. This basic description covers a large range of scheduling problems that has been widely studied. We propose to list some well known shop problems that have been studied in the literature:

JOB-SHOP PROBLEM
The job-shop problem consists in $n$ jobs and $m$ unary resources. Each job contains $m$ ordered activities associated to the $m$ different unary resource. The sequence of ordered activities and their durations are different inside each job. The objective function is to minimize the makespan. An example of job-shop problem could be the scheduling of an assembly line with products with slightly different options.

CUMULATIVE JOB-SHOP PROBLEM
The cumulative job-shop problem is a variant of the job-shop problem where resources are cumulative and activities can have different heights.

OPEN-SHOP PROBLEM
The open-shop problem is similar to the job-shop problem except that activities within a job are not ordered. An example of open-shop problem could be a set of workers having to realize a set of tasks during the day, each requiring a different single user work station.

FLOW-SHOP PROBLEM
The flow-shop problem is a restriction of the job-shop problem in which the sequence of activities (and thus of unary resource used) is the same in each job. An example of flow-shop could be an assembly line where products are built step by step on a conveyor belt.

NO-WAIT JOB-SHOP PROBLEM
The no-wait job-shop problem is a job-shop in which no time can elapse between activities within the same job. An example of no-wait job-shop problem could be the scheduling of products on an assembly line needing different layers of paint. To avoid the complete drying of the paint, the different layers have to be applied immediately one after the other.

PREEMPTIVE JOB-SHOP PROBLEM
The preemptive job-shop problem is a variant of the job-shop problem in which activities can be interrupted and continued later. An example of preemptive job-shop the scheduling of an assembly line

where a product can be removed from the production process to be completed later on.

GROUP-SHOP PROBLEM
In a group-shop problem, the order between activities belonging to the same job is only partly determined. Jobs are divided into sub-jobs. Activities within the same sub-job can be executed in any order (but overlapping is not allowed). Sub-jobs within the same job are ordered; meaning no activity from a given sub-job can start before all those of the preceding sub-job have ended. The group-shop problem is a generalization of the job-shop and the open-shop problems where sub-jobs are respectively all the activities in a job or single activities.

FLEXIBLE JOB-SHOP PROBLEM
In flexible job-shop problem, each activity is not associated to a single resource but it must use one from a defined set of unary resources.

### 2.2.2 *Resource Constrained Project Scheduling Problems (RCPSPs)*

This category of scheduling problems is defined as a set of activities partially ordered by precedence constraints. Activities are non-preemptive, have a defined duration and use a defined amount of cumulative resources. The objective function is to minimize either the makespan, the weighted lateness of the last activity, or the weighted sum of earliness and tardiness. The RCPSP can be seen as a the category of scheduling problems with non-preemptive activities and implying resources where we do not consider a set of jobs or the same types of resources as for shop problems. Many variants of the RCPSP exist. Such variants are obtained by adding reservoir resources, considering new precedence constraints, etc. For more details on RCPSPs, refer to [Bru99, RC15, KS97].

### 2.2.3 *Other Scheduling Problem*

Many other classes of scheduling problems exist and have been studied. They imply the mix of various resources, preemptive and non-preemptive activities, multiple optimization objectives, etc. As some real-world problems tend to lead to complex scheduling models, this mix of scheduling problems can often imply a wide range of different constraints and parameters. We do not intend to address all existing

problems in this thesis but the interested reader could find examples of scheduling problems that are neither shop problems nor RCPSPs in [Pin12].

## 2.3 A SCHEDULING PROBLEM EXAMPLE

In this section we consider a small scheduling problem example. Let us consider three workers, each having to build a different piece of furniture. To build a piece of furniture, each worker proceeds in three steps:

1. Cut the planks at the required dimensions.
2. Sand the planks such that they fit easily into each other.
3. Assemble the planks into the final piece of furniture.

To cut the planks, workers need to use the sawing station. This station can only be used by a single worker at a time. Similarly, the sanding of planks requires a sanding station that can also be used by a single worker at a time. The assembly of planks together does not require any specific working station. The goal is to obtain the three pieces of furniture as soon as possible as they have to be shipped together urgently to a client.

We can model this problem as a scheduling problem. For each worker, we define a job containing three activities: cutting, sanding and assembling. Activities inside the jobs are ordered by precedences (cutting comes before sanding which comes before assembling). The different activities and their durations are reported in Figure 2.7. The cutting and sanding activities both require a different working station that can be used by a single worker at a time. Hence, these two working stations are modeled as unary resources. As the objective is to produce the three pieces of furniture as soon as possible, we can model this as a makespan minimization. This problem can be thought of as a job-shop in which the last activity of each job does not use a unary resource. A feasible solution for this problem is shown in Figure 2.8.

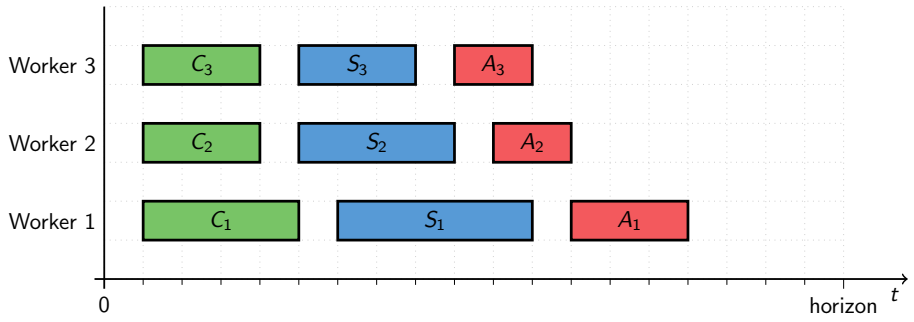



Figure 2.7: The activities grouped by job for the furniture building problem. This picture represents only the durations of the different activities and is not a solution to the problem. The 'C', 'S' and 'A' represent respectively the cutting, sanding and assembling activities. The indices associated to the capital letters corresponds to the associated worker.

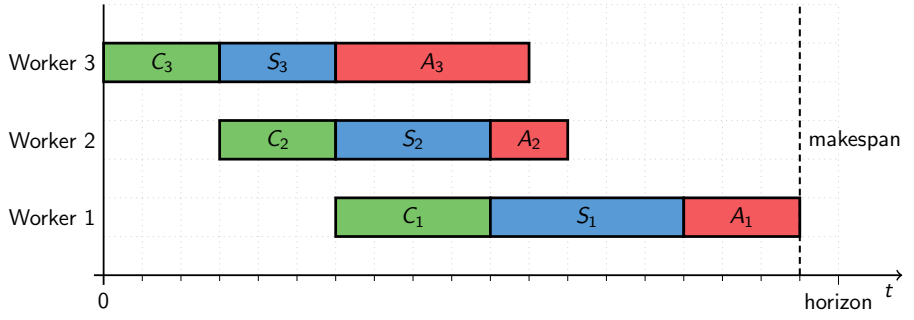



Figure 2.8: A feasible solution for the furniture building problem from Figure 2.7. Note that the cutting activities do not overlap each other in time as they share a unary resource. The same situation occurs for the sanding activities. However, assembling activities $A_2$ and $A_3$ can overlap as they do not share a unary resource. The makespan shown on the figure corresponds to the maximal ending time of the activities in the schedule.

# 3

# COMPARING PROPAGATORS WITH PERFORMANCE PROFILES

*I wanna be the very best, like no one ever was.*

—Pokemon Theme Song

*The Dark Side of the Moon has flash - the true flash that comes from the excellence of a superb performance.*

—Alan Parsons

*Don't lower your expectations to meet your performance. Raise your level of performance to meet your expectations.*

—Ralph Marston

*The only thing you owe the public is a good performance.*

—Humphrey Bogart

*Those that are most slow in making a promise are the most faithful in the performance of it.*

—Jean-Jacques Rousseau

Comparing the potential of various propagators associated to a constraint can be tricky. Most of the time, a propagator can be characterized by two attributes: its worst time complexity and the achieved pruning, i.e., the set of values removed from domains of variables in the scope of the constraint. When two propagators offer the same pruning, the one with the lowest time complexity is preferred. However, it happens that two propagators achieve different pruning, i.e., the set of values removed from domains of variables in the scope of the constraint differs between propagators. In some cases, some propagators achieve more pruning than others, but at the cost of a higher time complexity. Whether or not the additional pruning is worth the increased propagation time might depend on the size and type of the considered instance. It is thus important to design a comparison procedure to evaluate the performance of propagators with respect to each other.

This chapter describes the state-of-the-art propagator comparison method introduced by Sascha Van Cauwelaert et al. in [CLS15]. This approach is used in this thesis to compare the propagation procedures from Chapters 4 and 5 with current state-of-the-art propagators. The approach from Van Cauwelaert defines several solvers, based on a common model to which one propagator (a different one for each solver) is added. These models are run on a set of instances such that each run explores the same region of the search space. Various measures on performance metrics are performed on these runs. Each measure for a model on an instance represents the performance of the model on this particular instance for the considered performance metric. Then, to aggregate performances of solvers on all instances, performance profiles are used.

In this chapter, Section 3.1 describes the method used to ensure models associated to the various propagators explore the same region of the search space. Then Section 3.2 describes how the results can be aggregated in a single graphical representation to clearly visualize the performances achieved on a set of instances.

## 3.1 COMPARISON OF DIFFERENT MODELS

We consider $n$ propagators $\phi_1, \ldots, \phi_n$ whose performances have to be compared. These propagators are filtering algorithms for a constraint $c$ with a scope $\text{scope}(c) = \{x_y, \ldots, x_z\}$. Each propagator behaves as a filtering function mapping the domains of variables in this scope $\{D(x_y), \ldots, D(x_z)\}$ to a set of reduced domains $\{D'(x_y), \ldots, D'(x_z)\}$

for these variables. To compare the different approaches, [CLS15] proposes to consider a baseline model $M$ and all the models obtained by adding one of the propagators mentioned earlier: $M \cup \phi_k$ where $1 \leq k \leq n$. These different models will be run on a hopefully representative set of instances. Each run will provide a measure for each performance metric, e.g., the number of nodes browsed, the number of backtracks performed, the time of the run, etc. In order to obtain a meaningful comparison, for a given instance, the runs of the model $M$ and all other models $M \cup \phi_k$ should respect two conditions:

1. All runs explore the same region of the search space. Should different regions of the search space be visited by the different models, there would be no way to compare them.

2. The search nodes are visited by the different runs in the same order. Should this ordering differ for the different models, some might be advantaged because of the objective bounding.

Based on these two conditions, [CLS15] proposes a *replay strategy* instance by instance. For a given instance, it first runs the baseline model $M$ using a branching strategy $b$ until a timeout is reached or the search space has been completely visited. During the run of $M$, the sequence of visited search nodes is saved. Then, for each model $M \cup \phi_k$, the same sequence of search nodes is visited, skipping the nodes removed by the propagation of $\phi_k$. The measure on the considered performance metric is taken during each of these replay runs. This replay strategy allows to simulate the use of complex branching strategies while enforcing that the same region of the search space is visited. Algorithm 3.1.1 represents this replay strategy. In line 1, the first run of $M$ with branching $b$ is performed on the instance to obtain the sequence of nodes visited. Then, in line 2, the measure $p^i_{\text{baseline}}$ is obtained by replaying the sequence of nodes `nodeSequence`. Similarly, for all the propagators $\phi_k$, the measures $p^i_k$ are obtained by replaying the sequence of nodes using the corresponding model $M \cup \phi_k$.

The replay strategy from Algorithm 3.1.1 is used on every instance. This means that measures on performance metrics have been obtained for all the models $M \cup \phi_k$ on all instances. Now remains to define a way to aggregate these measures such that it provides information on whether a propagator performs *globally* better than another one, on the set of considered instances, on the different performance metrics. In the next section, performance profiles are explained as a way to achieve such objective.

**Algorithm 3.1.1 :** Replay Strategy

**Input** : $b$ The branching strategy for the run of $M$
**Input** : $i$ The instance considered
**Input** : $t_{\max}$ The max time of the baseline first run

```
nodeSequence ← Run(M, b, i)
p^i_baseline ← Replay(M, nodeSequence)
foreach k in 1, ..., n do
    p^i_k ← Replay(M ∪ φ_k, nodeSequence)
end
```

## 3.2 PERFORMANCE PROFILES

The performance profiles technique [DM02] aggregates the measures of a performance metric on a set of instances $\mathcal{I}$. The performance profile of a given performance metric $\tau$ is a cumulative distribution function $\rho(\tau)$. In the case where propagators are compared, the $\tau$ value represents the ratio between the measures for performance metric $p$ of model $M \cup \phi_k$ and $M$. Considering the measure of a performance metric $p_k^i$ of the model $M \cup \phi_k$ obtained by the replay strategy from Section 3.1 on instance $i \in \mathcal{I}$, the performance ratio $r_k^i$ is defined as follows:

$$r_k^i = \frac{p_k^i}{p_{\text{baseline}}^i}$$

From these ratios, the cumulative performance profile is defined as follows:

$$\rho_k(\tau) = \frac{1}{|\mathcal{I}|} \left| \left\{ i \in \mathcal{I} : r_k^i \leq \tau \right\} \right| \tag{3.1}$$

$\rho_k(\tau)$ is the proportion of instances for which the model $M \cup \phi_k$ had a performance ratio of *at most* $\tau$ in comparison to the baseline. For example, if the performance metric is the time needed by a propagator to reach a given search node, $\rho_k(0.5)$ represents the proportion of instances for which the model $M \cup \phi_k$ spent at most half the time needed by the baseline model to reach a given search node. Following this definition, models with a larger $\rho_k(\tau)$ for a given $\tau$ are to be preferred. The performance profiles can be represented graphically as curves in a 2D plot where the horizontal axis is $\tau$ and the vertical axis is $\rho(\tau)$. The sequence of points associated to a curve for a performance metric in a

2D performance profile plot is obtained with Algorithm 3.2.1. It begins by sorting the ratios in increasing order. Then, the sequence of points is iteratively built such that the ratio at position $j$ in the sorted ratios, $\texttt{sortedRatios}_j$, gives the point $\left(\texttt{sortedRatios}_j, \frac{j}{|\mathcal{I}|}\right)$. In our case, the

**Algorithm 3.2.1 :** Performance Profile 2D

**Input** : $r_k$ The ratios obtained by $M \cup \phi_k$ on $\mathcal{I}$
**Output** : pointSequence The sequence of points for the 2D curve

```
1 pointSequence ← {}
2 sortedRatios ← Sort(r_k)
3 for j in 1, ..., |I| do
4 |   point ← (sortedRatios_j, j/|I|)
5 |   Append(pointSequence, point)
6 end
7 return pointSequence
```

graphical representation obtained for the baseline model will always be a step function where a single step of height 1 happens at 1 on the horizontal axis. The percentage of instances for which a model $M \cup \phi_k$ obtains better or equivalent performances than the baseline model $M$ is given by $\rho_k(1)$, corresponding to the intersection of 2D curve of the model with the one of the baseline occurring in $(1, \rho_k(1))$.

As mentioned in [DM02], performance profiles offer many advantages over other presentations of data obtained by benchmarks. Many benchmarking results are published under the form of tables. Interpretation of results from these tables can be difficult, especially when the set of instances is quite large, hence leading to large tables. Similarly, reporting the average or cumulative total of a given performance metric can be problematic. Indeed, a small number of difficult instances (or too easy instances) can dominate the results. Another problem with reporting averages or cumulative totals is that it requires to drop instances for which any of the models failed, inducing a bias against the most robust models. Performance profiles do not suffer the drawbacks mentioned above. The use of performance ratios avoids a small number of difficult instances to dominate the results. Presenting a cumulative representation of these ratios allows to easily observe the trend of the results, especially since the graphical representation is concise and easy to read.

### 3.2.1 *Example*

Let us consider a small example where two propagation procedures, $\phi_1$ and $\phi_2$ are compared with regards to a baseline procedure. The comparison takes thus place over the models $M$, $M \cup \phi_1$ and $M \cup \phi_2$. In this example, the performance metric compared is the time taken by the different models to browse a given sequence of nodes obtained with the replay strategy presented in Section 3.1. The example measures are reported in Table 3.1.

| Instance | $M$ | $M \cup \phi_1$ | $M \cup \phi_2$ |
|---|---|---|---|
| $i_1$ | 5 | 6 | 7 |
| $i_2$ | 22 | 14 | 18 |
| $i_3$ | 12 | 14 | 8 |
| $i_4$ | 4 | 2 | 8 |
| $i_5$ | 62 | 10 | 30 |

Table 3.1: Example of time results obtained on the set of instances $\mathcal{I} = \{i_1, i_2, i_3, i_4, i_5\}$ by the solvers compared.

| Instance | $M$ | $M \cup \phi_1$ | $M \cup \phi_2$ |
|---|---|---|---|
| $i_1$ | 1.00 | 1.20 | 1.40 |
| $i_2$ | 1.00 | 0.64 | 0.82 |
| $i_3$ | 1.00 | 1.67 | 0.67 |
| $i_4$ | 1.00 | 0.50 | 2.00 |
| $i_5$ | 1.00 | 0.16 | 0.48 |

Table 3.2: Performance ratios of measures from Table 3.1.

From these measures, the performance ratios are reported in Table 3.2. By sorting the ratios per solver and creating 2D points with the equations $\left(\texttt{sortedRatios}_j, \frac{j}{|\mathcal{I}|}\right)$, the points reported in Table 3.3 are obtained.

Finally, Figure 3.1 is the graphical representation of the performance profiles with the points obtained in Table 3.3. Note that in our perfor-

| | | | | | |
|---|---|---|---|---|---|
| $M$ | (1.00, 0.2) | (1.00, 0.4) | (1.00, 0.6) | (1.00, 0.8) | (1.00, 1.0) |
| $M \cup \phi_1$ | (0.16, 0.2) | (0.50, 0.4) | (0.64, 0.6) | (1.20, 0.8) | (1.67, 1.0) |
| $M \cup \phi_2$ | (0.48, 0.2) | (0.67, 0.4) | (0.82, 0.6) | (1.40, 0.8) | (2.00, 1.0) |

Table 3.3: Performance ratios of measures from Table 3.1.

mance profile representations, when a curves disappears, it is because it is below another one superposing it. When two performance profiles superpose each other, they share the same performance values (but only on the superposed parts). This figure highlights the fact that the approach with $\phi_1$ performs generally better than the approach with $\phi_2$. Indeed, its curve is above the one of $M \cup \phi_2$, meaning that it performs better for any proportion of instance. One can also observe that both $M \cup \phi_1$ and $M \cup \phi_2$ perform better than the baseline model $M$ for at least 60% of the instances. In this example, $M \cup \phi_1$ is at worst 1.67 times slower than the baseline model $M$.

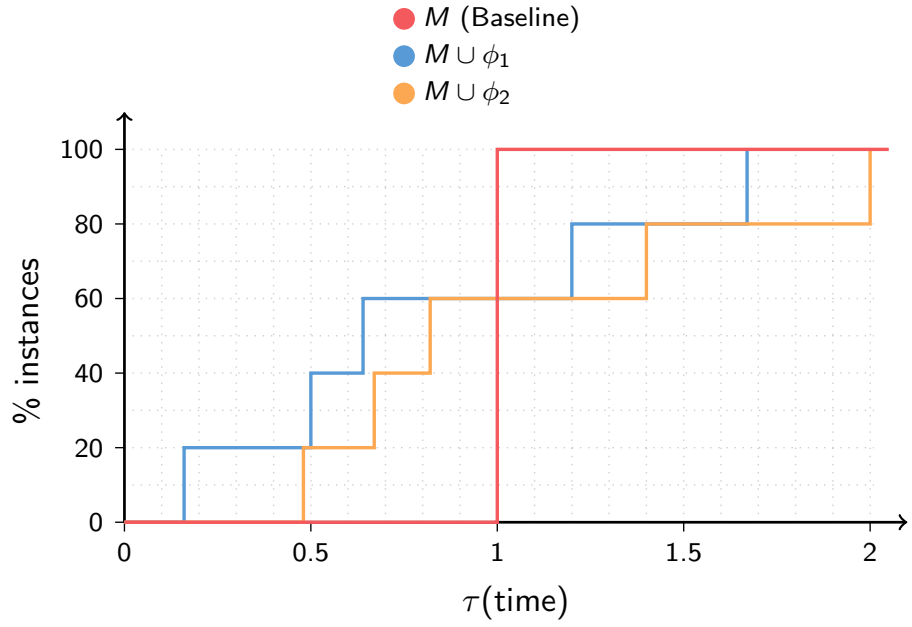



Figure 3.1: Graphical representation of performance profiles for measures from Table 3.1.

# Part II

# ENHANCING PROPAGATION IN SCHEDULING

# 4

# FORWARD CHECKING PROPAGATION FOR THE NESTED GCC

*Progress is not an illusion, it happens, but it is slow and invariably disappointing.*

—George Orwell

*An error does not become truth by reason of multiplied propagation, nor does truth become error because nobody sees it.*

—Mahatma Gandhi

*Why live? Life was its own answer. Life was the propagation of more life and the living of as good a life as possible.*

—Ray Bradbury, *The Martian Chronicle*

*Woodstock was both a peaceful protest and a global celebration.*

—Richie Havens

*Men are mortal. So are ideas. An idea needs propagation as much as a plant needs watering. Otherwise both will wither and die.*

—B. R. Ambedkar

The Global Cardinality Constraint (GCC) from Régin [Rég96] is a core constraint of the CP framework. The GCC imposes bounds on the number of times a value can be assigned to a set of variables. Many real world applications contain constraints that can be modeled with GCC combinations. As such, several variants and extensions of GCC have been studied in the literature. The Nested Global Cardinality Constraint (Nested GCC) is a generalization of the GCC. Nested GCC imposes bounds on the number of times a value can be assigned in several *nested* sets of variables. This chapter presents a new Forward Checking propagation procedure for the Nested GCC constraint. The Nested GCC constraint is used in an application context described in Chapter 8. It models stock and order book constraints over a production plan.

There are two main contributions for the propagation of Nested GCC in FWC in this chapter. First, we propose a pre-computation procedure that allows to strengthen the bounds provided as argument to obtain stronger and faster propagation using a decomposition of multiple FWC GCC propagators. The second contribution is the implementation of a dedicated global Nested GCC FWC propagator that allows propagation with a reduced worst time complexity in comparison to a decomposition of multiple FWC GCC propagators.

This chapter starts by a definition of the Nested GCC, introduced by Zanarini and Pesant in Section 4.1. Then, Section 4.2 brings to light the need of dedicated propagation procedures to overcome the pruning missed by a classic GCC propagator decomposition. It also gives a brief description of the GAC propagation procedure from [ZP07]. Section 4.3 presents a pre-computation procedure to strengthen the bounds provided to the constraint in order to achieve more important pruning. It then proposes a dedicated FWC propagator for the Nested GCC with a reduced time complexity and compares its performances with other existing propagators. Finally, Section 4.4 evaluates the performances of the various propagation procedures mentioned and introduced in this chapter.

## RELATED PUBLICATIONS

[Dej+16] Cyrille Dejemeppe, Olivier Devolder, Victor Lecomte, and Pierre Schaus. “Forward-Checking Filtering for Nested Cardinality Constraints: Application to an Energy Cost-Aware Production Planning Problem for Tissue Manufac-

turing." In: *Integration of AI and OR Techniques in Constraint Programming*. Banff, Canada: Springer International Publishing, 2016.

## 4.1 THE CONSTRAINT

The Nested GCC constrains a set of variables $X = \{x_1, \ldots, x_n\}$ that can be separated into $p$ disjoint sets. Let $\{X^k\}$ with $1 \leq k \leq p$ represent these disjoint sets of variables. Each set contains variables with successive indices $X^k = \{x_{k_{\min}}, \ldots, x_{k_{\max}}\}$ such that $k_{\min}$ is the index coming right after $(k-1)_{\max}$ and $k_{\max}$ is the index coming just before $(k+1)_{\min}$. We define the following nested unions of these sets:

$$\chi^k = \bigcup_{1 \leq j \leq k} X^j$$

The sets $\chi^k$ can be seen as ranges containing variables from $x_1$ to $x_{k_{\max}}$. This means that $\chi^p$ contains all the variables from $X$, covering variables from $x_1$ to $x_{p_{\max}} = x_n$. As these unions are nested, the following property holds:

$$\chi^1 \subseteq \chi^2 \subseteq \ldots \subseteq \chi^p$$

To each variable $x_i \in X^k$ corresponds its domain $D(x_i)$. The union of domains of the variable set $X^k$ is written $D(X^k)$ and the union of domains over $X$ is simply written $D(X)$. The number of occurrences of a value $v \in D(X)$ in a subset $\chi^k$ is bounded between a lower bound $l_v^k$ and an upper bound $u_v^k$. Formally, the signature of the Nested GCC is:

$$\texttt{Nested GCC}\left(\left[X^1, \ldots, X^p\right], \left[l^1, \ldots, l^p\right], \left[u^1, \ldots, u^p\right]\right)$$

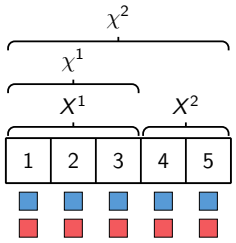
where $l^i$ and $u^i$ $(1 \leq i \leq p)$ are respectively lower and upper bound arrays containing the lower and upper bounds for each possible value $v \in D(X)$. The constraint implied by such signature is as follows:

$$\forall k \in [1, p], \forall v \in D(X) : l_v^k \leq \left|\left\{x \in \chi^k \mid x \to v\right\}\right| \leq u_v^k$$

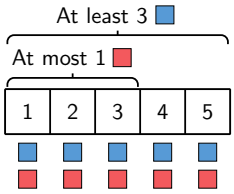
where $x \to v$ expresses that variable $x$ is assigned to value $v$. To ease the reading in the rest of this chapter, we refer to ranges $[1, t]$ to denote the range of variables from $x_1$ to $x_t$.

*Example*

Let us consider a small set of five variables $X$ separated in two disjoint sets: $X^1 = \{x_1, x_2, x_3\}$ and $X^2 = \{x_4, x_5\}$. These five variables have initial domains that contains the same two elements: $D^{\text{init}}(X) = \{\text{blue}, \text{red}\}$. This example is represented in Figure 4.1a. Let us now consider that we have the following upper bound: $u^1_{red} = 1$, and the following lower bound $l^2_{blue} = 3$. These bounds are represented in Figure 4.1b. Concretely, it constrains that at most one variable from the set $\{x_1, x_2, x_3\}$ is bound to value *red* and at least 3 variables from the set $\{x_1, x_2, x_3, x_4, x_5\}$ are bound to value *blue*.



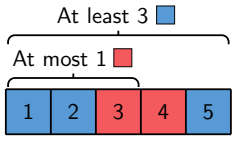

(a) Disjoint sets of variables



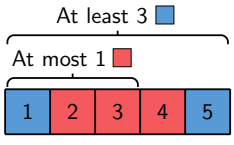

(b) Lower and upper bounds

Figure 4.1: Small example of the Nested GCC constraint. Indexed squares represent variables and the small colored squares below each variable represent the values remaining in their domains.

Figure 4.2a shows a feasible solution for the example from Figure 4.1 satisfying the Nested GCC constraint. Figure 4.2a shows an unfeasible solution for this example. Indeed, $u^1_{red}$ is exceeded: there are 2 variables assigned to red in $\chi^1$. Furthermore, $l^2_{blue}$ is not reached since there are only 2 variables assigned to blue in $\chi^2$.




(a) Feasible solution




(b) Unfeasible solution

Figure 4.2: Two potential solutions for example from Figure 4.1.

## 4.2 GAC PROPAGATION FOR NESTED GCC

The Nested GCC presented earlier can be modeled with a combination of classic GCCs. Considering a Nested GCC applied on a set of variables $X$ split into $k$ sets as defined in Section 4.1, a single classic GCC could be applied to each subset of variables $\chi^k$ with bounds $l^k$ and $u^k$. However, as proven in [ZP07], the propagation obtained would not be able to achieve Global Arc Consistency (GAC).

Let us consider the example from Figure 4.3. With classic multi-GCCs decomposition, there would be two GCC propagators used: the first one constrains variables in the range $\{x_1, x_{11}\}$ to contain at least four variables assigned to red; the second one constrains variables in the range $\{x_1, x_{16}\}$ to contain at most seven variables assigned to red. None of these two propagators would detect a failure on this small example. However, there are already four variables assigned to value red in the range $[1, 16]$ constrained to contain at most seven variables assigned to red. This imposes that the range before the first variable assigned to value red (range from 1 to 12) should contain at most three variables assigned to value red. This is in conflict with the lower bound stating that there should be at least four variables assigned to red in the range $[1, 11]$. This brings to light the need of a dedicated propagator for Nested GCC to achieve GAC.

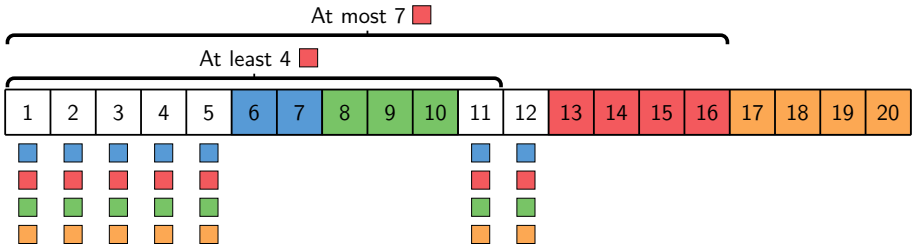



Figure 4.3: Example of pruning missed with classic GAC GCCs decomposition for a Nested GCC constraint.

A GAC propagator for Nested GCC is introduced in [ZP07]. It proposes a graph representation $G(V, E)$ for the Nested GCC constraint. This graph contains a node for every variable $x_i \in X$ as well as a source $s$ and a sink $t$. Additionally, for every subset $X^k$, and every possible value in $D(X)$, a node is added. The arcs have a lower and an upper capacity and connect the nodes as follows:

- For each variable $x_i$ there is an arc from the source $s$ to $x_i$ with capacity $[1, 1]$.
- For each variable $x_i \in X^k$ there is an arc with capacity $[0, 1]$ from the variable to each value node of $X^k$.
- For each subset $X^k$, there is an arc from each value node to the value node of $X^{k+1}$ corresponding to the same value with capacity $[l^k, u^k]$.
- For each value node of $X^p$, there is an arc from this node to the sink $t$ with capacity $[l^p, u^p]$.

The small example from Figure 4.1 would be represented with the graph shown in Figure 4.4.

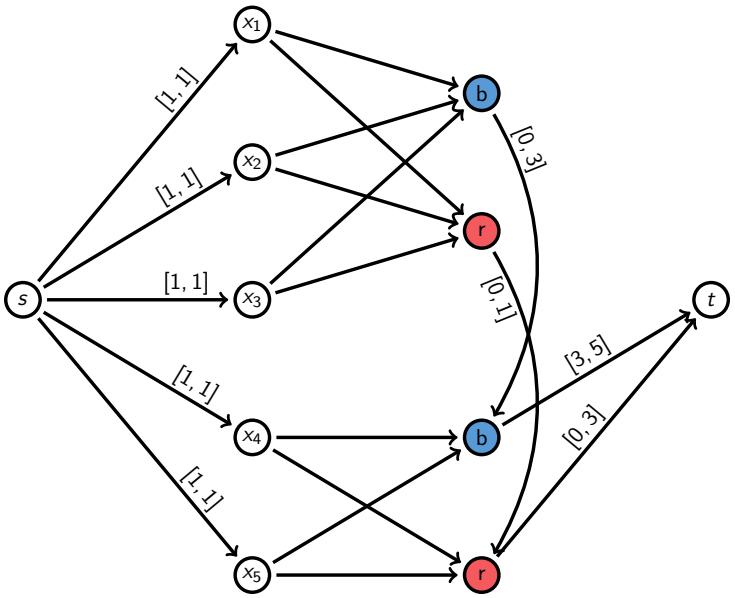



Figure 4.4: Graph representation of the small example from Figure 4.1. Arcs with no bounds displayed take the default values $[0, 1]$.

The GAC propagator for Nested GCC introduced in [ZP07] proposes to use this graph representation. A feasible flow in this graph represents a feasible assignment for the Nested GCC. A unary flow between a variable $x_i$ and a value node $v$ corresponds to the assignment $x_i \to v$. Similarly to the GAC propagator for classic GCC [Rég96], the filtering algorithm finds a feasible flow in the graph to check the feasibility of the Nested GCC. When there is no feasible flow, the constraint

is unfeasible. If a feasible flow exists, pruning can be performed. First, the residual graph of the flow found is built. Then, all the strongly connected components of the residual graph are computed. Finally, all the arcs that do not belong to any strongly connected component can be removed. Every arc from a node $x_i$ to a value node $v$ that has been removed corresponds to the removal of $v$ from $D(x_i)$.

Considering the graph representation $G(V, E)$, the GAC propagator for Nested GCC has a time complexity $\mathcal{O}(|V||E|)$ to find a feasible flow and $\mathcal{O}(|V| + |E|)$ to find unfeasible values. For these graphs the number of nodes $|V|$ is in $\mathcal{O}(n + k|D(X)|)$ and the number of edges $E$ is in $\mathcal{O}(n|D(X)| + k|D(X)|)$.

## 4.3 A NESTED GCC FORWARD CONSISTENT PROPAGATOR

As explained in [Smi05], maintaining a higher level of consistency takes more time; on the other hand, if more values can be removed from the domains of variables, the search effort will be reduced and this will save time. Whether or not the time saved by the values removed outweighs the time spent on propagation depends on the problem. In practice, many solvers (such as the very efficient OR-Tools [OR-10]) use a default Forward Checking filtering (FWC) for the GCC. For this reason, we propose to develop a new FWC propagation procedure for the Nested GCC.

In this section, we design a FWC propagator achieving both a potentially stronger and faster pruning when compared to a naive decomposition of $k$ Forward Checking GCC propagators (FWC-GCCs). The improvement in pruning is obtained by a pre-processing step that strengthens the bounds of the cardinalities $l_v^k$ and $u_v^k$. For simplicity, in this section we suppose that the set of variables $X = \{x_1, \ldots, x_n\}$ is separated into singletons $X^k = \{x_k\}$. As such, $l_v^k$ is a lower bound for the number of occurrences of $v$ in the range $[1, k]$ (thus for variables $\{x_1, \ldots, x_k\}$). The improvement in terms of running time is obtained by maintaining incremental counters avoiding the need to propagate every sub-GCC on each domain update. We present first the pre-computation step, then the dedicated FWC propagator.

### 4.3.1 *Stronger Bounds Pre-Computation*

This step aims at tightening the bounds $l_v^k$ and $u_v^k$ specified in the signature of the constraint and minimizing the number of these to a minimal set. Two reasonings can be done:

1. Between different ranges for the same value (e.g. the occurrences of red in range $[1,4]$ and range $[1,5]$).
2. Between the bounds for the different values specified for the same range $[1,t]$ (e.g. the occurrences of red versus blue in range $[1,6]$).

The first one corresponds to per-value deductions and the second one refers to inter-value deductions. Once both sets of deductions have been performed, it is possible to reduce the bounds obtained to a minimal set of bounds containing all the information needed to apply pruning.

*Per-Value Deductions*

Per-value deduction is applied for each value in two sweeping of the ranges: a forward update followed by a backward update. Intuitively, the following forward and backward deductions can be made:

LOWER BOUNDS
If there are at least *two* red in range $[1,4]$, then there are at least *two* red in range $[1,5]$ (forward), and at least *one* red in range $[1,3]$ (backward).

UPPER BOUNDS
If there are at most *two* red in range $[1,4]$, then there are at most *three* red in range $[1,5]$ (forward), and at most *two* red in range $[1,3]$ (backward).

We can make those deductions based on the quantities $l_v^t$ and $u_v^t$ containing respectively the best-known lower and upper bounds on the occurrences of $v$ for range $[1,t]$. This is done by traversing these values for each range once forward and once backward. The forward update of these values is defined as follows, $t$ increasing from 2 to $n$:

$$l_v^t = \max \begin{cases} l_v^t \\ l_v^{t-1} \end{cases} \qquad u_v^t = \min \begin{cases} u_v^t \\ u_v^{t-1} + 1 \end{cases}$$

Similarly, the backward update is defined as follows, $v$ decreasing from $n-1$ to 1:

$$l_v^t = \max \begin{cases} l_v^t \\ l_v^{t+1} - 1 \end{cases} \qquad u_v^t = \min \begin{cases} u_v^t \\ u_v^{t+1} \end{cases}$$

An example of per-value deduction for lower bounds for a given value $v$ is shown in Figure 4.5. Initial lower bounds (black dots) in the gray zone are updated since dominated by the other specified bounds. The arrays displayed in this example represent the quantities $l_v^t$ at the different steps of the bound tightening. **Original** represents the original bounds specified as argument, **Filled** represents the bounds after application of the forward update (left to right in the array) in Figure 4.5a and after backward update (right to left in the array) in Figure 4.5b. A similar example of per-value deduction for upper bounds for a given value $v$ is shown in Figure 4.6.

*Inter-Value Deductions*

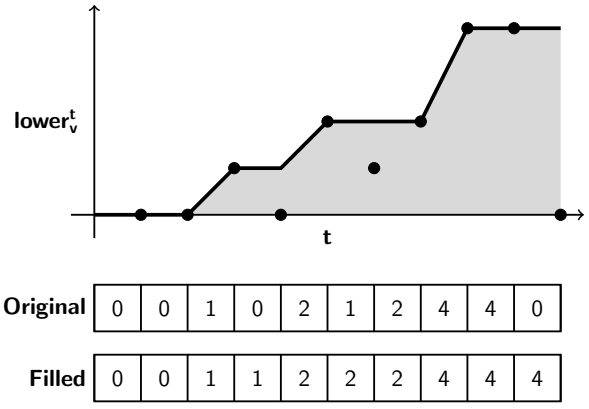
Inter-Value deduction is applied on each range and updates the bound of a value with regards to the bounds of other values. Intuitively, for a given range $[1, t]$ and for some value $v$, the following deductions can be made:

LOWER BOUNDS
On a given range, if the upper bound of a value $u_v^t$ is small, then the lower bounds of other values in $[1, t]$ are large.

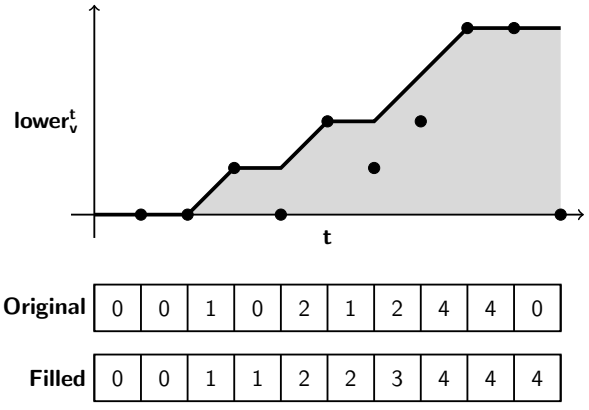
UPPER BOUNDS
On a given range, if the lower bound of a value $l_v^t$ is large, then the upper bounds of other values in $[1, t]$ are small.

For example, let us consider the bounds of a value $v$ on the range $[1, 5]$. If the sum of the lower bounds on the other values is 3 ( i.e., if $\sum_{w \neq v} l_w^5 = 3$), then there can be *at most* 2 occurrences of $v$ on this range. This implies that $u_v^5 \leq 2$. Similarly if the sum of the upper bounds for other values is 3 ($\sum_{w \neq v} u_w^5 = 3$), then there must be *at least* 2 occurrences of $v$ on this range. This implies that $l_v^5 \geq 2$.

We can make those deductions based on the quantities $l_v^t$ and $u_v^t$ containing respectively the best-known lower and upper bounds on




(a) After the forward update




(b) After the backward update

Figure 4.5: Example of per-value deduction for lower bounds of a value $v$.

the occurrences of $v$ for range $[1, t]$. For every value $v$ and every range $[1, t]$ bounds are updated as follows:

$$l_v^t = \max \begin{cases} l_v^t \\ t - \sum_{w \neq v} u_w^t \end{cases} \qquad u_v^t = \min \begin{cases} u_v^t \\ t - \sum_{w \neq v} l_w^t \end{cases}$$

*Reduction to a Minimal Set of Bounds*

After the tightening step of the bounds $l_v^t$ and $u_v^t$, the number of these bounds can be minimized to only keep the useful bounds in a decomposition of the Nested GCC. If a bound for a given range is on a plateau i.e., it is equal to bounds on previous and next range, it does

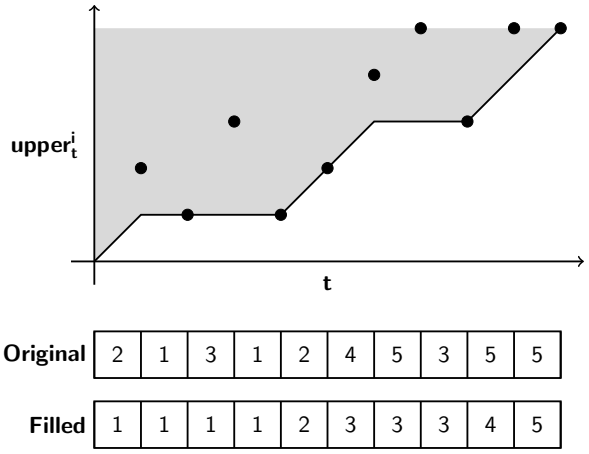



Figure 4.6: Example of per-value deduction for upper bounds of a value $v$.

not contain any useful information. Another situation where a bound does not carry information for a given range happens when it is in the middle of a slope i.e., it is larger/smaller than bound on previous range and smaller/larger than bound on next range. Formally, only lower bounds meeting the two following conditions carry useful information:

$$\text{keep } l_v^t \text{ if } \begin{cases} l_v^t = l_v^{t-1} + 1 \\ l_v^t = l_v^{t+1} \end{cases}$$

A similar reasoning can be applied to filter the upper bounds. Formally, the upper bounds carry useful information if the two following conditions are met:

$$\text{keep } u_v^t \text{ if } \begin{cases} u_v^t = u_v^{t-1} \\ u_v^t = u_v^{t+1} - 1 \end{cases}$$

On the example of Figure 4.7, the minimal set of useful bounds is circled in the graph and those are given in the **Filtered** array. A similar example to deduce the minimal set of upper bounds for a given value is shown in Figure 4.8.

The pre-computation step is done only once, at the initialization of the constraint. The final minimal set of bounds obtained after 1) the per-value deductions, 2) inter-value deductions and 3) minimization of the set of bounds, is the unique smallest set of bounds that contains all the useful information initially specified by the quantities $l_v^t$ and $u_v^t$.

Furthermore, the set of ranges on which those final bounds apply is always a subset of the ranges at which a lower and upper bounds were originally given.

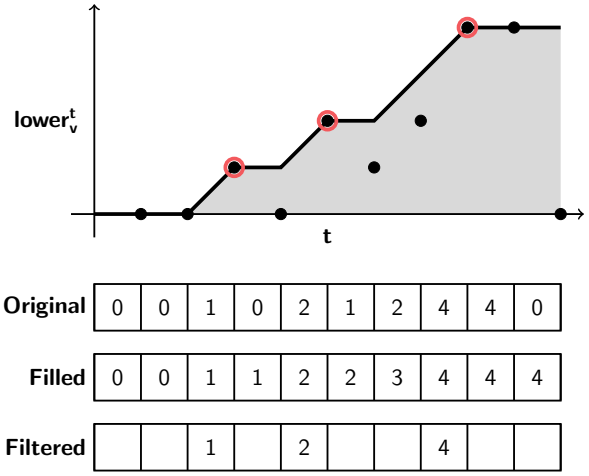



Figure 4.7: Reduction to the minimal set of useful lower bounds from Figure 4.5.

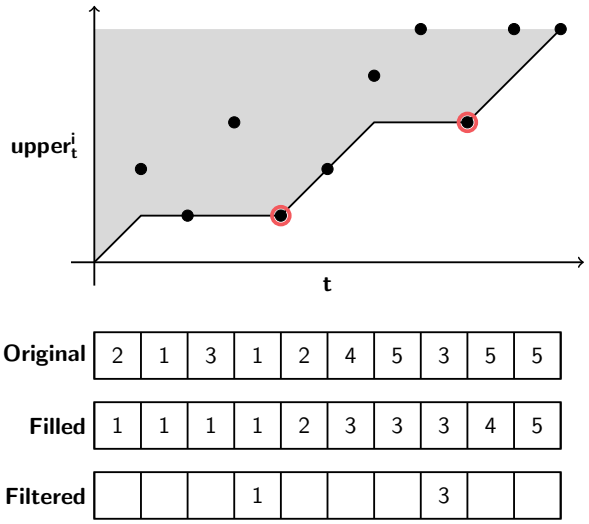



Figure 4.8: Reduction to the minimal set of useful upper bounds from Figure 4.6.

A simplified version of the code used in OscaR to perform stronger bound deduction is provided in Appendix A.3.

*Bounds Optimality*

The three steps we have described here to obtain stronger bounds have the two strong properties. We provide here an intuition of the reasons why these two properties hold.

BOUNDS OPTIMALITY

Bounds are optimal in the sense that running a second time these steps would not provide better bounds. The per-value deduction step has to be performed first since it will also *fill* each period with a bound for each value (some values might not be bounded on several prefixes in the original bounds). Hence, performing the inter-value deductions before the per-value deductions would result in less tight bounds. Repeating per-value deductions after the inter-value deduction step will not bring any further improvement. Indeed, in the case were some bound, either lower or upper, has been improved by the inter-value deductions, this new bound will apply on a range that was previously between two *critical ranges* for this value (i.e., a critical range is a range for which the bound (either lower or upper) is not in the middle of a slope). Hence two cases can appear:

1. It dominates one or both of the two critical bounds. In such case, the dominated bound(s) will only be removed by the per-value deductions, but no further deductions will be deduced.
2. It is dominated by these two critical bounds. In such case, this dominated bound will only be removed by the per-value deductions, but no further deductions will be deduced.

From this we deduce that only one run of first the per-value then the inter-value deductions is needed. Finally, the bound reduction step has to go last. Indeed, this step removes bounds that are not critical i.e., bounds that influence nor per-value, nor inter-value deductions.

MINIMAL SET OF BOUNDS

The final bounds are a subset of the original ones. Indeed, the only bounds that are kept are the critical bounds. In the initial bounds, only a subset of them are critical bounds. Then, using the same reasoning as the one used earlier, when a bound is determined to be stronger, it either dominates one/both of its surrounding bounds or it is dominated by its two surrounding bounds; the dominated bounds are dropped. The number of critical bounds can thus only remain the same or decrease, but can never increase. The cardinality of the set of

initial bounds is thus an upper bound on the number of bounds that remains after the three steps of stronger bounds pre-computation.

### 4.3.2 *A dedicated FWC propagator for the Nested GCC*

With the stronger bounds computed above, a decomposition of $p$ FWC-GCCs propagators, one propagator per constrained range, achieves stronger pruning. However, its amortized time complexity remains in $\mathcal{O}(p)$ per domain update since all $p$ propagators running in $O(1)$ are potentially triggered. We present here a propagator that runs in $\mathcal{O}(\log(p))$ amortized time and offers the same pruning. As a reference point, the pruning given by FWC-GCCs is such that

- when the number of variables whose domain still contains a given value decreases to the lower bound associated to it, these variables are assigned to the value.
- when the number of variables bound to a given value increases to the upper bound associated to it, this value is removed from all other variables.

The main challenge of this algorithm is to avoid checking those variable counts on every lower or upper bound when an update is received. In order to do that, for every value that we track and for both lower and upper bounds, we divide the variables into the segments that are formed by the bounds, and only count variables inside those segments. For example, if we have a maximum of 2 red in range $[1, 3]$ and a maximum of 5 red in range $[1, 8]$, we will separate the variables into the segments $[1, 3]$ (the first 3 variables) and $[4, 8]$ (the next 5 variables). We justify in the next paragraphs why local checks inside those segments are enough to detect and trigger the required pruning. This example is shown in Figure 4.9.

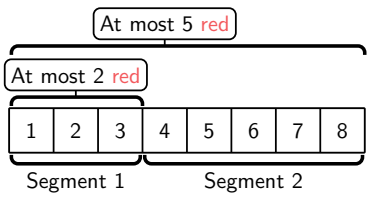



Figure 4.9: Example of segment decomposition.

Let us examine the differences between the bounds in our example: $5 - 2 = 3$. We will call this difference the *critical point* of the segment $[4, 8]$. If the number of variables bound to red in that segment reaches 3, then there will be at least 3 occurrences of red in that segment. As a consequence, if the pruning condition in range $[1, 3]$ is met, so that we have 2 variables bound to red in $[1, 3]$, then in total there will be at least 5 variables bound to red in the range $[1, 8]$, so we have to prune there as well. In other words, pruning in $[1, 3]$ can only happen if pruning in $[1, 8]$ also happens; and since in both cases pruning means removing the value red from all unbound variables, it becomes useless to track the upper bound on $[1, 3]$.

Conversely, if there are *less than* 3 variables bound to red in the segment $[4, 8]$, then pruning for the upper bound of range $[1, 8]$ will happen strictly after pruning happens in $[1, 3]$ (if ever). Indeed, pruning in $[1, 3]$ happens when 2 variables in that segment are bound to red, and at that point less than 5 variables would be bound to red in $[1, 8]$.

For the leftmost segment, since there is no bound on the left, we simply define the critical point as the bound on the right, in this case 2 for segment $[1, 3]$. In this segment, reaching the critical point by having 2 variables bound to red means reaching a pruning case, so we have to remove the red value from the last variable. If the number of variables bound to red is strictly under the critical point, however, no pruning can be performed.

From these remarks we can notice that no pruning will happen in a segment until it reaches its critical point. All that is left is to precisely determine what to do when it is reached. We have taken upper bounds as an example, but the critical point also makes sense for lower bounds: instead of counting the number of variables bound to the value, we count the number of variables that have the value in their domain.

We can also observe a useful property of critical points: if we combine two consecutive segments, the distance to the critical point in the merged segment will be the sum of the distances to the critical points in the small segments. Indeed, when summing the critical points, the middle bound will cancel itself out; and the number of variables that are bound to a value or that have a value in their domain is clearly the sum of those numbers in the segments that are being merged.

*Pruning cases and segment merging*

Let us now develop an update strategy based on critical points. We split the set of variables into contiguous disjoint segments as described above. In the leftmost segment, pruning can happen only when its corresponding critical point has been reached. For other segments, if their respective critical points have not been reached, then no pruning can occur before some pruning happens on the left bound. When the critical point of a segment is reached, we can consider two different actions to perform, depending on whether the considered segment is the leftmost one or not.

First, if the segment is the leftmost segment, we have to trigger pruning in it. As none of the segments on its right has reached its critical point, no pruning should occur on those. Once the pruning has been applied to the leftmost segment, it is removed and its neighboring right segment, if it exists, is marked as the leftmost segment. To achieve fast pruning, we propose to maintain a list of unbound variables still containing a particular value in an array based reversible doubly linked list. This allows value removal in constant time (as there is one list per possible value). We refer to this list as the *unbound list*. When a critical point is reached, the pruning on a segment will only be applied on variables in the unbound list.

Second, if the segment is *not* the leftmost segment, then reaching the critical point makes the bound on the left of the segment completely redundant in terms of pruning with the bound on the right of the segment. Therefore, the bound on the left can be forgotten, and this segment can be merged with its left neighboring segment. Since distance to the critical point is additive, the larger segment will not have reached its critical point either. To keep the propagator efficient in terms of time complexity, we have to determine efficiently to which segment a variable belongs. We also have to determine an efficient way to merge segments. This problem can be solved easily using a union-find data structure [Tar75].

Our propagation procedure will be triggered by two different events:

1. A variable $x_i$ has been bound to a value $v$ and it is inside an upper bound segment.

2. A value $v$ has been removed from a variable $x_i$ and it is inside a lower bound segment.

When the first event occurs, Algorithm 4.3.1 is performed. It first removes the variable $x_i$ from the unbound list associated to value $v$. Then, by using a `Find` operation (on our union-find data structure), it will fetch the segment containing the variable. The counter of variables bound to $v$ in the segment is augmented. If this counter has reached the critical point, then the actions to take depend on whether the segment is the leftmost segment or not. If it is the leftmost segment, the value $v$ is removed from all the unbound variables in the segment (this could trigger a Failure). Then, the segment on the right is marked as the leftmost segment (if it exists). On the other hand, if the segment for which the critical point has been reached is not the leftmost segment, it is merged with the segment on its left (using a `Union` operation on our union-find data structure).

**Algorithm 4.3.1 :** Update Upper Bound Segment

**Input** : $x_i$ The variable that has been bound
**Input** : $v$ The value to which the variable has been assigned

```
1  Remove(unboundList_v, x_i)
2  segment ← Find(v, x_i)
3  counter_v(segment) ← counter_v(segment) + 1
4  if counter_v(segment) = critical point then
5      if segment is leftmost segment then
6          foreach x_j in segment do
7              RemoveValue(x_j, v)
8          end
9          if Right(segment) exists then
10             Right(segment) becomes leftmost segment
11         end
12     else
13         Union(Left(segment), segment)
14     end
15 end
```

Similarly, Algorithm 4.3.2 is performed when a value $v$ has been removed from a variable $x_i$ that is inside a lower bound segment. The operations performed are similar to those from Algorithm 4.3.1 except that the counter represents the number of variables in the segment whose domain does not contain $v$. Furthermore, if the critical point of the segment is reached and the segment is the leftmost segment, all

unbound variables in the segment are assigned to $v$ (This could lead to a Failure).

**Algorithm 4.3.2 :** Update Lower Bound Segment

**Input** : $x_i$ The variable whose domain has been reduced
**Input** : $v$ The value that was removed from the variable domain

```
1  Remove(unboundList_v, x_i)
2  segment ← Find(x_i)
3  counter_v(segment) ← counter_v(segment) - 1
4  if counter_v(segment) = critical point then
5  |  if segment is leftmost segment then
6  |  |  foreach x_j in segment do
7  |  |  |  Assign(x_j, v)
8  |  |  end
9  |  |  if Right(segment) exists then
10 |  |  |  Right(segment) becomes leftmost segment
11 |  |  end
12 |  else
13 |  |  Union(Left(segment), segment)
14 |  end
15 end
```

*Time Complexity*

The complexity analysis assumes one has access to the $\Delta$ change of the variables as for instance proposed in [SM+13] for the OscaR solver [Osc12] also available in OR-Tools [OR-10], or the advisors of Gecode [LS07].

Let us define $u$ as the number of updates, that is, the sum of the number of value removals over the whole search. Note that when the constraint itself removes a value from a variable, it counts in $u$ as well. We will also use $n$, the number of variables, and $p$, which as earlier is the number of distinct ranges involved in the bounds. Looking at the steps performed when a value has been either removed or assigned, we can deduce the time complexity for a particular update. Note that even though the loops on lines 6-9 in Algorithms 4.3.1 and 4.3.2 can take $\mathcal{O}(n)$ for one particular update to be processed, the variables pruned also count as updates, so it remains amortized constant time per update.

The worst-time complexities of Algorithm 4.3.1 and Algorithm 4.3.2 are $\mathcal{O}(\log(p))$. Indeed, the operation `Remove` operation is performed on the doubly linked list, `UnboundList` and is thus performed in $\mathcal{O}(1)$. When we combine all the other operations performed at each update, we discover that the total time complexity is the number of updates multiplied by the cost of a union-find operation. As mentioned in [SW11], there are implementations of the union-find data structure where both operations `Union` and `Find` can be performed in $\mathcal{O}(\alpha(p))$. In such implementations, $\alpha(p)$ is the inverse of an Ackermann function [Sun71] $A(p,p)$ growing extremely fast, leading $\alpha(p)$ to be constant in amortized running time. However, as this is implemented in a CP framework, we are working with a reversible union-find structure. As such, a particular update could be repeated arbitrarily many times in different places in the search tree. This means we cannot use the amortized $\mathcal{O}(\alpha(p))$ complexity for union-find operations, but rather the $\mathcal{O}(\log(p))$ worst case. As a result, we obtain a time complexity in $\mathcal{O}(u\log(p))$ for the whole search, or an amortized time complexity of $\mathcal{O}(\log(p))$ per update.

*FWC and GAC Pruning*

The pruning achieved by either a decomposition into multiple FWC GCC propagators or the dedicated FWC Nested GCC propagator described earlier is larger when using the stronger bounds previously pre-computed. However, this stronger pruning is still smaller than the one obtained by the GAC graph-based propagation proposed by Zanarini and Pesant [ZP07]. We illustrate this with the small example shown in Figure 4.10. In this example, there are 6 variables whose domains contain 2 different values: red and blue. The initial bounds lower and upper bounds constraining variables from $x_1$ to $x_t$ to be assigned respectively at least $l_v^t$ and at most $u_v^t$ times to value $v$ are:

$$\begin{cases} l^3_{red} = 2 \\ l^3_{blue} = 0 \\ l^6_{red} = 0 \\ l^6_{blue} = 0 \end{cases} \quad \begin{cases} u^3_{red} = 3 \\ u^3_{blue} = 3 \\ u^6_{red} = 3 \\ u^6_{blue} = 6 \end{cases}$$

The pre-computation steps that we have described earlier compute the following set of stronger bounds:

$$\begin{cases} l^3_{red} = 2 \\ l^3_{blue} = 0 \\ l^6_{red} = 2 \\ l^6_{blue} = 3 \end{cases} \quad \begin{cases} u^3_{red} = 3 \\ u^3_{blue} = 1 \\ u^6_{red} = 3 \\ u^6_{blue} = 4 \end{cases}$$

Despite these stronger bounds, the propagation achieved by any of the FWC Nested GCC propagation procedure that we have described is not able to detect the failure in our example. There is a failure because since variables $x_5$ and $x_6$ are already bound to value red, the bound $u^6_{red} = 3$ imposes that there can be at most one variable in the range $x_1, \ldots, x_4$ bound to red. This is in contradiction with the bound $l^3_{red} = 2$ that imposes at least two variables in the range $x_1, \ldots, x_3$ to take the value red. However, the GAC graph-based propagation procedure, whose propagation graph is illustrated in Figure 4.11, detects this failure. Again, while GAC propagation is stronger than the procedure that we have described, its worst case time complexity is much larger than the fast FWC Nested GCC propagation procedure..

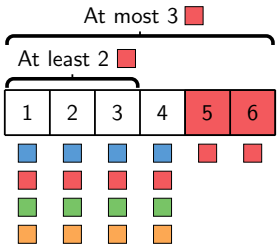



Figure 4.10: Example where FWC Nested GCC propagation misses a failure that is however detected by the GAC Nested GCC propagation from [ZP07].

## 4.4 EXPERIMENTAL RESULTS

To evaluate our propagator, we used the OscaR solver [Osc12] and ran instances on AMD Opteron processors (2.7 GHz). For each considered instance, we used the three following filtering procedures for the Nested GCC constraint:

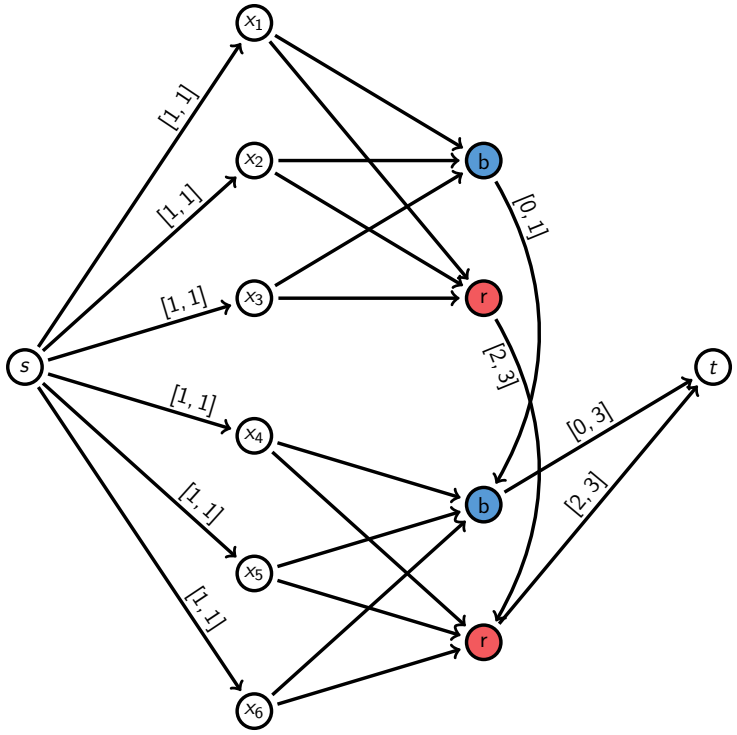



Figure 4.11: Prop[agation graph used by GAC Nested GCC propagation from [ZP07] for the example from Figure 4.10.

1. Decomposition into multiple GCC FWC propagators with the original bounds ($\phi_{\text{GCCFWCs}}$); there is one GCC FWC propagator for each range constrained by the original bounds.
2. Decomposition into multiple GCC FWC propagators with the strengthened bounds from Section 4.3.1 ($\phi_{\text{PrecompGCCFWCs}}$); there is one GCC FWC propagator for each range constrained by the strengthened bounds.
3. The Nested GCC FWC propagator with $\mathcal{O}(\log(p))$ time complexity from Section 4.3.2 ($\phi_{\text{NestedGCCFWC}}$). This version also uses the stronger bounds after the pre-computation step.

### 4.4.1 *Considered benchmarks*

We have considered instances for a very simple model. This model considers a given set of $n$ periods corresponding to a set of variables $X$ with the same initial domain. Several cardinality constraints hold

on $p$ ranges going from period 1 to period $k$ ($1 \leq k \leq n$). These cardinality constraints are represented with a Nested GCC constraint. To each period $i$ is associated a random cost $c_i$. To each possible value $v$ is associated a random multiplier $w_v$; these multipliers are stored in a vector $W$. The objective considered is the minimization of the total cost (obtained by multiplying the cost of each period by the multiplier of the value assigned to the period's variable). This objective has been added to avoid a too large number of solutions. Formally, the model is as follows:

$$\begin{aligned} &\text{minimize} && \sum_{i=1}^{n} c_i \cdot \texttt{Element}(x_i, W) \\ &\text{such that} && \texttt{Nested\_GCC}\left(\left[X^1, \ldots, X^p\right], \left[l^1, \ldots, l^p\right], \left[u^1, \ldots, u^p\right]\right) \end{aligned}$$

where the `Element` constraint is used to obtain the multiplier associated to the value of the variable at period $i$.

We have constructed random instances for the model described earlier with the following parameters:

NUMBER OF VARIABLES
: The different numbers of variables will allow to test the efficiency of the propagators on both small and large ranges of variables. We have considered the following numbers of variables: 20, 50, 100, 200, 400.

NUMBER OF VALUES
: The number of possible values for each variable, i.e., the cardinality of the domains. We have considered the following numbers of values: 3, 5, 10, 15, 20, 40.

NUMBER OF BOUNDS
: The number of bounds imposed. This number of bounds is common to all the values. For each bound, the range, value and whether it is a min or a max bound is computed randomly. The following number of bounds were chosen: 20, 50, 100, 150, 200, 400, 800.

For each of these configurations, we have generated 25 different instances. We have applied two filters on these instances:

1. The unfeasible instances were not considered. Indeed, instances that were unfeasible were most of the time detected at the root node (or at a very small depth in the search tree); leaving us with very small propagation time and null or tiny number of backtracks.

2. The instances that were closed in under one second by the slowest propagation procedure were discarded. The measures of time and number of backtracks are too small to be relevant in this case.

From the $5 \cdot 6 \cdot 7 \cdot 25 = 5,250$ generated instances, only 281 remained after the filters had been applied. Indeed, it is quite complicated to generate feasible instances with a large density of bounds (i.e., many ranges are constrained on different values). The results presented in this section are those obtained on these 281 instances.

### 4.4.2 *Comparison of the Three Models*

In order to present fair results regarding the benefits that are provided by the considered propagators, we have followed the methodology introduced in [CLS15]. An overview of this methodology is provided in Chapter 3. In brief, the approach presented in [CLS15] proposes to pre-compute a search tree using the filtering that prunes the less - the *baseline* propagator - and then to *replay* this search tree using the different studied filtering procedures. We used $\phi_{\text{GCCFWCs}}$ as the baseline filtering, and the *binary static* search strategy to construct the search tree. The search tree construction time was limited to 120 seconds. We then constructed *performance profiles* as described in [CLS15]. A concise description of the use of performance profiles to compare propagators is provided in Chapter 3.

#### *Performance Profiles on Number of Backtracks*

The performance profiles for backtracks are illustrated in Figure 4.12. There is a single performance profile representing $\phi_{\text{PrecompGCCFWCs}}$ and $\phi_{\text{NestedGCCFWC}}$ as they offer the same pruning. Indeed, both propagators are based on the same improved bounds offering the same additional pruning.

The pruning achieved by either $\phi_{\text{PrecompGCCFWCs}}$ or $\phi_{\text{NestedGCCFWC}}$ is much more important than the one achieved by $\phi_{\text{GCCFWCs}}$. This illustrates that the pre-computation of stronger bounds as explained in Section 4.3.1 allows not only to reduce the set of considered bounds, but also to achieve stronger pruning for FWC propagation. For only 5% of the instances is the pruning equivalent between FWC propagation based on the original bounds or on stronger pre-computed bounds. Furthermore, when some additional pruning is achieved with the help

of stronger bounds, it can be substantial: for at least 43% of the instances, the number of backtracks is reduced by at least a factor two.

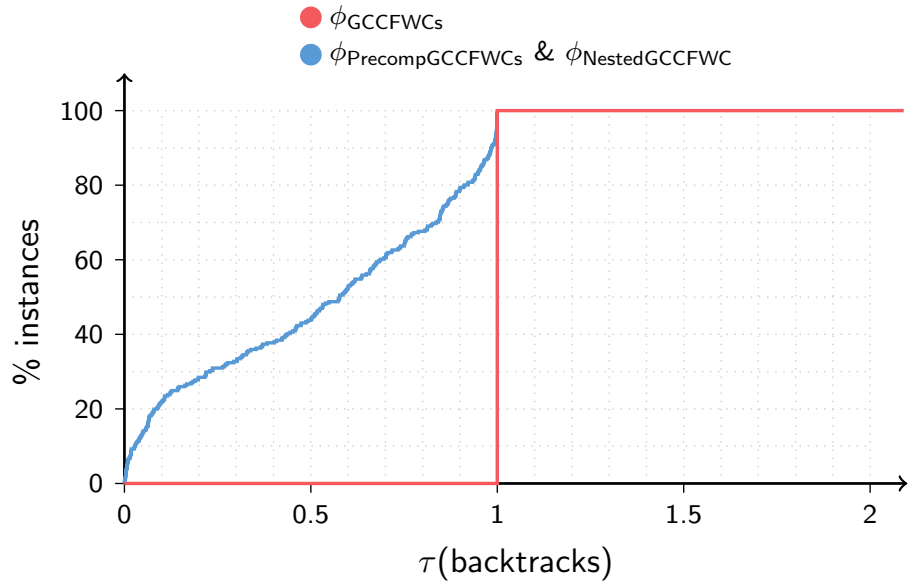



Figure 4.12: Performance profiles of the three models for the number of backtracks.

*Performance Profiles on Resolution Time*

The performance profiles for replay time of sequences of nodes generated for 120 seconds with $\phi_{\text{GCCFWCs}}$ and the binary static search heuristic are illustrated in Figure 4.13. The difference in terms of time between $\phi_{\text{NestedGCCFWC}}$ and $\phi_{\text{PrecompGCCFWCs}}$ is not very large. One could expect a larger difference in terms of propagation time due to the different time complexities: $\mathcal{O}(\log(p))$ for $\phi_{\text{NestedGCCFWC}}$ and $\mathcal{O}(p)$ for $\phi_{\text{PrecompGCCFWCs}}$. This can be explained by the multiplying constant of the time complexity associated to $\phi_{\text{NestedGCCFWC}}$. Indeed, even though the complexity is reduced from $\mathcal{O}(p)$ to $\mathcal{O}(\log(p))$, it can only be done by the addition of a reversible data structure that adds a small overhead to the operations performed on a single update. Furthermore, one update will only trigger the propagation of the GCC FWC propagators applying on a range containing the variable concerned by the update. Nevertheless, there is a small difference between $\phi_{\text{NestedGCCFWC}}$ and $\phi_{\text{PrecompGCCFWCs}}$. As this difference exists, one

should consider the implementation of our Nested GCC FWC propagator for problems where the Nested GCC constraint is the bottleneck of the resolution in terms of time. This can be detected by measuring the time required to propagate the Nested GCC constraint during the whole resolution of an instance and then comparing it to the total time taken to solve it.

On the other hand, the time difference between $\phi_{\text{GCCFWCs}}$ and both $\phi_{\text{PrecompGCCFWCs}}$ and $\phi_{\text{NestedGCCFWC}}$ is important. This is directly related to the additional pruning performed by the latter propagation procedures over the former. As $\phi_{\text{PrecompGCCFWCs}}$ and $\phi_{\text{GCCFWCs}}$ share the same time complexity in $\mathcal{O}(p)$, the smaller amount of nodes the former has to go through allows it to be faster. Furthermore, the number of bounds considered by $\phi_{\text{PrecompGCCFWCs}}$ may be smaller than the number of original bounds, leading to less GCC FWCs propagators. For at least 73% of the instances, $\phi_{\text{PrecompGCCFWCs}}$ is at least twice as fast as $\phi_{\text{GCCFWCs}}$. In the case of $\phi_{\text{NestedGCCFWC}}$, it is at least twice as fast as $\phi_{\text{GCCFWCs}}$ on at least 77% of the instances.

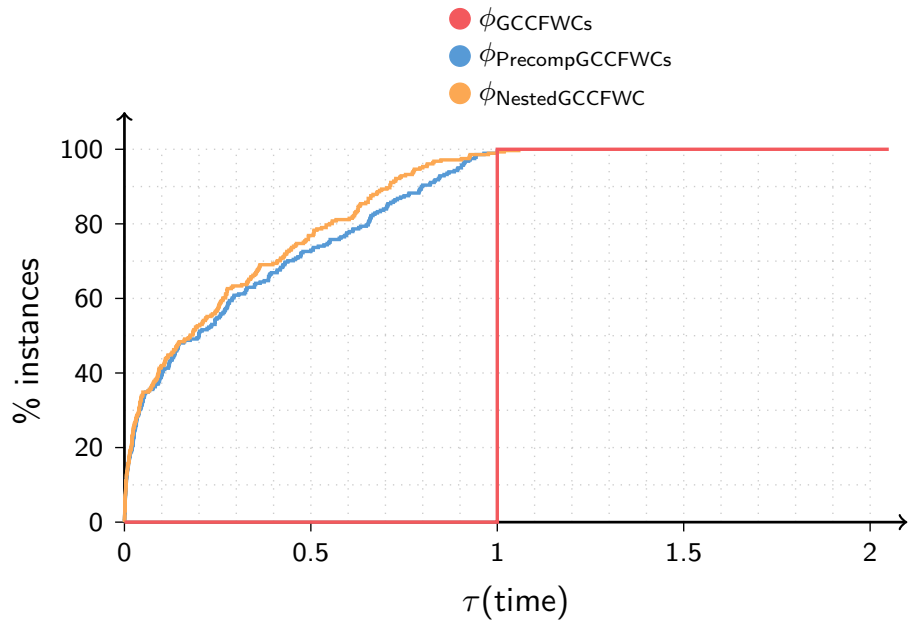



Figure 4.13: Performance profiles of the three models for the replay time on a sequence of nodes as explained in Chapter 3.

## CONCLUSION

In this chapter, we have discussed the FWC propagation of the Nested GCC constraint from [ZP07]. The FWC propagation of this constraint can be performed using a decomposition into multiple GCC FWCs propagators. The first improvement we have proposed to this approach is a pre-computation procedure that can obtain new bounds stronger than the original ones. This pre-computation procedure proceeds in three successive steps. The first step performs per-value deductions and considers the different ranges constrained for a single value. The second step performs inter-value deduction that applies on all the bounds on values occurring over the same range. Finally, the third step reduces the number of bounds obtained to a minimal set of useful bounds. This set of stronger bounds is always smaller or equivalent, in terms of size, to the original set of bounds. All the steps from this pre-computation procedure are light in terms of both computation time and memory used.

The pre-computed stronger bounds can be used with a decomposition into multiple classic GCC FWCs; there will be one GCC FWC propagator for each of the constrained range in the set of stronger bounds. The pruning achieved by this decomposition on the improved bounds is potentially larger than the one achieved on the original set of bounds. Furthermore, as this new set of bounds is potentially smaller than the original one, it should allow faster propagation time even when no additional pruning can be observed. Our experiments performed on a large set of instances has proven that the gain brought by such pre-computation in terms of pruning is substantial. Furthermore, the classic GCC FWC decomposition using stronger bounds has proven to be much faster than a version using the original weaker bounds.

The second contribution of this chapter is the introduction of a dedicated Nested GCC FWC propagator. This propagator performs the same pruning than a decomposition into multiple GCC FWC propagators. However, it has two advantages over this decomposition. First, it is a global propagator; hence it allows a more concise declaration of the model. Second, it has a reduced time complexity in $\mathcal{O}(\log(p))$ instead of $\mathcal{O}(p)$ and is therefore faster. However, our experiments have shown that the gain in terms of time is relatively small in comparison to a decomposition into multiple GCC FWC propagators using pre-computed stronger bounds. Hence, as the new propagator introduced demands

some consequent work in terms of implementation, we would advise the reader to implement that dedicated Nested GCC FWC propagator only if its performances are critical for the considered problem.

# 5

# THE UNARY RESOURCE WITH TRANSITION TIMES

*Welcome my son, welcome to the machine.*

—Pink Floyd, *Welcome to the Machine*

*Life is really simple, but we insist on making it complicated.*

—Confucius

*Life is pleasant. Death is peaceful. It's the transition that's troublesome.*

—Isaac Asimov

*Come on, Rory! It isn't rocket science, it's just quantum physics!*

—The Doctor, *Doctor Who*

*Fools ignore complexity. Pragmatists suffer it. Some can avoid it. Geniuses remove it.*

—Alan Perlis

The unary resource constraint [Vil04a] is a core constraint of scheduling problems. It imposes that two activities using the same unary resource cannot overlap in time. Many problems also impose that activities needing the same unary resource must be distant by a minimal amount of time. If the amount of time between two activities depends on the two activities between which it takes place, the amount of time is said to be sequence-dependent. This constraint is referred to in the literature as *sequence-dependent transition times*[1] [Elm68].

This chapter presents extensions of the classic unary resource propagation algorithms from [Vil07, Vil04a, Vil04b, VBČ05] that include propagation over *sequence-dependent* transition times between activities. A wide range of real-world scheduling problems from the industry involves transition times between activities. An example is the quay crane scheduling problem in container terminals [Zam+13] where the crane is modeled as a unary resource and transition times represent the moves of the crane on the rail to go from one position to another along the vessel to load/unload containers.

There are several contributions for the propagation of the Unary Resource with transition times in this chapter. First, we have extended thet Θ-tree and Θ-Λ-tree data structures from Vilím such that they include transition times in the computation of lower bounds for the earliest completion time of a set of activities. Then, we show how to use these structures in modified versions of the four propagation algorithms from [Vil07] (Overload Checking, Detectable Precedences, Not-First/Not-Last and Edge Finding) to obtain global propagators for the unary resource with transition times.

Section 5.1, gives an overview of the tackled problems and of current state-of-the-art techniques to solve them. Then, Section 5.2 explains the requirements needed to integrate transition times propagation. Section 5.3 describes how to obtain lower bounds for the time spent by transitions between activities from a set. Then, Section 5.4 introduces the integration of this bound in extended Θ-tree structures to efficiently compute the ect of a set of activities. Section 5.5 then explains how classic unary algorithms can consider transition times by using the extended Θ-tree structures. Finally, the results obtained by the new propagation procedure are reported in Section 5.6.

1 Sometimes also referred to as sequence-dependent setup times.

## RELATED PUBLICATIONS

[DCS15] Cyrille Dejemeppe, Sascha Van Cauwelaert, and Pierre Schaus. "The Unary Resource with Transition Times." In: *Principles and Practice of Constraint Programming*. Cork, Ireland: Springer International Publishing, 2015.

## 5.1 THE CONSTRAINT

As explained in Chapter 2, a scheduling problem is modeled by associating three variables to each activity $A_i$: $s_i$, $e_i$, and $d_i$ representing respectively the starting time, ending time and duration of $A_i$. These variables are linked together by the following relation:

$$s_i + d_i = e_i$$

Depending on the considered problem, global constraints linking the activity variables are added to the model. In this work, we are interested in the unary resource constraint. A *unary resource*, sometimes referred to as a *machine*, is a resource allowing only a single activity to use it at any point in time. As such, all activities demanding the same unary resource cannot overlap in time; for every pair of activity on the resource, either one must precede the other:

$$\forall i, j : (A_i \ll A_j) \vee (A_j \ll A_i)$$

This constrains the variables as follows:

$$\forall i, j : (e_i \leq s_j) \vee (e_j \leq s_i)$$

The unary resource can be generalized by requiring transition times between activities. A transition time $\mathrm{tt}_{i,j}$ is a minimal amount of time that must occur between two activities $A_i$ and $A_j$ if $A_i \ll A_j$ (precedes). These transition times are described in a matrix $\mathcal{M}$ in which the entry at line $i$ and column $j$ represents the minimum transition time between $A_i$ and $A_j$: $\mathrm{tt}_{i,j}$. We assume that transition times respect the triangular inequality. That is, inserting an activity between two activities always increases the time between these activities:

$$\forall i, j, k \;\; i \neq j \neq k : \mathrm{tt}_{i,j} \leq \mathrm{tt}_{i,k} + \mathrm{tt}_{k,j}$$

The unary resource with transition times imposes the following relation:

$$\forall i, j : (e_i + \mathrm{tt}_{i,j} \leq s_j) \vee (e_j + \mathrm{tt}_{j,i} \leq s_i) \tag{5.1}$$

The triangular inequality assumption is mandatory for the computation of lower bounds of transitions in a set of activities, as described in Section 5.3. However, all the propagation algorithms described in Section 5.5 could run without this assumption, as long as they use lower bounds computed with techniques that do not need this property.

### 5.1.1 *Related Work*

As described in a recent survey [All+08], scheduling problems with transition times can be classified in different categories. First the activities can be grouped in *batches* (i.e., a machine allows several activities of the same batch to be processed simultaneously) or not. Transition times may exist between successive batches. A CP approach for batch problems with transition times is described in [Vil07]. Secondly the transition times may be *sequence-dependent* or *sequence-independent*. Transition times are said to be sequence-dependent if their durations depend on both activities between which they occur. On the other hand, transition times are sequence-independent if their duration only depends on the activity after which it takes place. The problem category we study in this chapter is non-batch sequence-dependent transition time problems.

Several methods have been proposed to solve such problems. Ant Colony Optimization (ACO) approaches were proposed in [GPG01, Tah+05] while [BSV08, ABF05, ORS10, GVV08] propose Local Search and Genetic Algorithm methods. [ORS10] introduces a propagation procedure with the Iterative Flattening Constraint-Based Local Search technique. Many contributions using CP approaches to solve sequence-dependent problems have been proposed.

Focacci et al [FLN00] introduce a propagator for job-shop problems involving alternative resources with non-batch sequence-dependent transition times. In this approach a successor model is used to compute lower-bounds on the total transition time. The filtering procedures are based on a minimum assignment algorithm (a well known lower bound for the Travelling Salesman Problem). In this approach the total transition time is a constrained variable involved in the objective function (the makespan).

In [ABF04], each resource is associated to a Travelling Salesman Problem with Time Window (TSPTW) relaxation. The activities used by a resource are represented as vertices in a graph and edges be-

tween vertices are weighted with corresponding transition times. The TSPTW obtained by adding time windows to vertices from bounds of corresponding activities is then resolved. If one of the TSPTW is found unsatisfiable, then the corresponding node of the search tree is pruned. A similar technique is used in [AF08] with additional propagation.

In [Wol09], an equivalent model of multi-resource scheduling problem is proposed to solve sequence-dependent transition times problems. Finally, in [GH10], a model with a reified constraint for transition times is associated to a specific search to solve job-shop with sequence-dependent transition times problems.

Carlier and Pinson [CP89] introduce a branching scheme called Immediate Selection that can be used to solve many disjunctive shop problems. They however do not consider transition times problems in their procedure. Baptiste and Lepape [Bap96] propose extensions of several job problems, including an extension dealing with transition times. However, they only mention the fact that the Edge Finding update rules remains applicable when transition times are involved; there is no mention of any algorithm allowing to apply those update rules with a small time complexity. Brucker and Thiele [BT96] also extend Carlier and Pinson's work to take transition times into account. They use a disjunctive graph allowing to maintain a lower bound on the makespan. This lower bound is obtained by considering, for all the machines of the problem the earliest completion time of the activities executed on it. Several relaxations and variations of this techniques are proposed to do so. This approach is however strongly coupled with a makespan minimization objective. Furthermore, the update of the bounds of activities is not determined using Detectable Precedences [Vil04a], Not-First/Not-Last [VBČ05] or Edge Finding [VBČ05].

To the best of our knowledge, the CP filtering introduced in this chapter is the first one proposing to extend *all* the classic filtering algorithms for unary resources (Overload Checking [BLN01], Detectable Precedences [Vil04a], Not-First/Not-Last [VBČ05] and Edge Finding [VBČ05]) by integrating transition times, *independently of the objective function* of the problem. This filtering can be used in any problem involving a unary resource with sequence-dependent transition times.

### 5.1.2 *Unary Resource Propagators in CP*

As mentioned in Chapter 2, the earliest starting time of an activity $A_i$ denoted $\text{est}_i$, is the time before which $A_i$ cannot start. The latest

starting time of $A_i$, $\text{lst}_i$, is the time after which $A_i$ cannot start. The domain of $s_i$ is thus the interval $[\text{est}_i; \text{lst}_i]$. Similarly the earliest completion time of $A_i$, $\text{ect}_i$, is the time before which $A_i$ cannot end and the latest completion time of $A_i$, $\text{lct}_i$, is the time after which $A_i$ cannot end. The domain of $e_i$ is thus the interval $[\text{ect}_i; \text{lct}_i]$. These definitions can be extended to a set of activities $\Omega$. For example, $\text{est}_\Omega$ is defined as follows:

$$\text{est}_\Omega = \min\{\text{est}_j \mid j \in \Omega\}$$

The propagation procedure for the unary resource constraint introduced in [Vil07] contains four different propagation algorithms all running with time complexity in $\mathcal{O}(n\log(n))$ : Overload Checking (OC), Detectable Precedences (DP), Not-First/Not-Last (NFNL) and Edge Finding (EF). Let us consider that all the activities using a given unary resource form the set $T$. These propagation algorithms all rely on an efficient computation of the earliest completion time of a set of activities $\Omega \subseteq T$ using data structures called *Theta Tree* and *Theta-Lambda Tree* introduced in [Vil07]. Our contribution is a tighter computation of the lower bound of $\text{ect}_\Omega$ taking into account the transition times between activities.

## 5.2 TRANSITION TIMES EXTENSION REQUIREMENTS

The propagation procedure we introduce in this chapter relies on the computation of $\text{ect}_\Omega$, the earliest completion time of a set of activities $\Omega \subseteq T$. It follows the idea introduced in [Vil07] that proposes filtering algorithms for the classic unary resource without transition times. These propagation algorithms rely on an efficient computation of a lower bound $\text{ect}^{LB0}_\Omega$ of the earliest completion time of a set of activities $\Omega \subseteq T$, defined as:

$$\text{ect}^{LB0}_\Omega = \max_{\Omega' \subseteq \Omega} \{\text{est}_{\Omega'} + d_{\Omega'}\} \tag{5.2}$$

This bound does not consider the transitions that might occur between the activities in $\Omega$. We propose a tighter lower bound that depends on the transition times occurring between activities inside $\Omega$. Let $\Pi_\Omega$ be the set of all possible permutations of activities in $\Omega$. For a given permutation $\pi \in \Pi_\Omega$ (where $\pi(i)$ is the activity taking place at

position $i$ in the permutation $\pi$), we can define the total time spent by transition times, $\mathrm{tt}_\pi$, as follows:

$$\mathrm{tt}_\pi = \sum_{i=1}^{|\Omega|-1} \mathrm{tt}_{\pi(i),\pi(i+1)}$$

A lower bound for the earliest completion time of $\Omega$ can then be defined as follows:

$$\mathrm{ect}_\Omega^{LB1} = \max_{\Omega' \subseteq \Omega} \left\{ \mathrm{est}_{\Omega'} + d_{\Omega'} + \min_{\pi \in \Pi_{\Omega'}} \mathrm{tt}_\pi \right\}$$

Unfortunately, computing this value is NP-hard. Indeed, computing the optimal permutation $\pi \in \Pi$ minimizing $\mathrm{tt}_\pi$ is equivalent to solving a TSP. Since embedding an exponential algorithm in a propagator is generally impractical, a looser lower bound can be used instead.

Our goal is to keep the overall $\mathcal{O}(n \log(n))$ worst time complexity of Vilím's algorithms. The lower bound $\underline{\mathrm{tt}}(\Omega')$ must therefore be available in constant time for a given set $\Omega'$. Our approach to obtain constant time lower-bounds for a given set $\Omega'$ during search is to base its computation solely on the cardinality $|\Omega'|$. More precisely, for each possible subset of cardinality $k \in \{0, \ldots, n\}$, we compute the smallest transition time permutation of size $k$ on the set $T$ of all activities requiring the resource:

$$\underline{\mathrm{tt}}(k) = \min_{\{\Omega' \subseteq T:\ |\Omega'|=k\}} \left\{ \min_{\pi \in \Pi_{\Omega'}} \mathrm{tt}_\pi \right\} \tag{5.3}$$

For each $k$, the lower bound computation thus requires one to find the shortest node-distinct $(k-1)$-edge path between any two nodes, which is also NP-hard as it can be casted into a Resource-Constrained Shortest Path Problem. Hopefully, we propose in later sections various lower bounds for this value that can be obtained in polynomial time.

Our final lower bound formula for the earliest completion time of a set of activities, making use of pre-computed lower-bounds of transition times, is:

$$\mathrm{ect}_\Omega^{LB2} = \max_{\Omega' \subseteq \Omega} \left\{ \mathrm{est}_{\Omega'} + d_{\Omega'} + \underline{\mathrm{tt}}(|\Omega'|) \right\} \tag{5.4}$$

The different lower bounds of $\mathrm{ect}_\Omega$ can be ordered as follows:

$$\mathrm{ect}_\Omega^{LB0} \leq \mathrm{ect}_\Omega^{LB2} \leq \mathrm{ect}_\Omega^{LB1} \leq \mathrm{ect}_\Omega \tag{5.5}$$

The next sections will detail how we have managed to compute $\mathrm{ect}_\Omega^{LB2}$ efficiently. This requires two properties. First, we need methods

to pre-compute tight $\underline{\text{tt}}(k)$ in polynomial time. We detail in Section 5.3 various lower bounds to achieve the pre-computation in polynomial time. Second, we need to maintain the computation of $\text{ect}_\Omega^{LB2}$ incremental. This means that adding or removing an activity to $\Omega$ should not require a complete $\text{ect}_\Omega^{LB2}$ re-computation.

## 5.3 TRANSITIONS IN A SET OF ACTIVITIES

The computation of $\underline{\text{tt}}(k)$ for all $k \in \{1, \ldots, n\}$ is NP-hard. This is a constrained shortest path problem (for $k = n$ it amounts to solving a TSP) in a graph where each node corresponds to an activity and directed edges between nodes represent the transition time between corresponding activities. Although these computations are achieved at the initialization of the constraint, we propose to use polynomial lower bounding procedures instead. Several approaches are used and since none of them is dominating *all the other ones*[2], we simply keep the maximum of the computed lower bounds.

### 5.3.1 *Sum of Minimal Transitions*

The first idea that comes to mind to compute $\underline{\text{tt}}(k)$ for $k \in [1, n]$ is to consider all the transition times of the problem and sum the $k-1$ minimal ones. This has been proposed by [BT96]. Let us consider the set of all transition times, $\mathcal{M}$. Most of the time, $\mathcal{M}$ is represented in a matrix form where the entry in line $i$ and column $j$ is $\text{tt}_{i,j}$. Let us consider an activity $A_i$. By definition, as the triangular inequality holds, there can be only a single transition time applied *from* $A_i$, a single $\text{tt}_{i,j}$ for any $j$. Therefore, an acceptable lower bound $\underline{\text{tt}}(k)$ is to consider the smallest transition time $\text{tt}_{i,j}$ for all activities $A_i$ and sum the $k$ minimal ones. Formally, the set of the lowest transitions *from* activities is defined as follows:

$$\text{tt}_{i\rightarrow} = \min_j \ \text{tt}_{i,j}$$
$$\mathcal{M}_\rightarrow = \{\text{tt}_{i\rightarrow} \mid 1 \leq i \leq n\}$$

2 There is however one relaxation that is dominated by another one, this will be mentioned as it is detailed.

To obtain the $k$ smallest elements from this set, we use the following recursive formula:

$$\begin{aligned}\mathcal{M}_{\rightarrow}^{0} &= \varnothing \\ \mathcal{M}_{\rightarrow}^{n} &= \mathcal{M}_{\rightarrow}^{n-1} \cup \left\{\min\ \left(\mathcal{M}_{\rightarrow} \setminus \mathcal{M}_{\rightarrow}^{n-1}\right)\right\}\end{aligned}$$

Similarly, there can be only one transition time applied *to* $A_i$, a single $\mathrm{tt}_{j,i}$ for any $j$. Therefore, the set of the lowest transitions *to* activities is defined as follows:

$$\begin{aligned}\mathrm{tt}_{i\leftarrow} &= \min_j\ \mathrm{tt}_{j,i} \\ \mathcal{M}_{\leftarrow} &= \{\mathrm{tt}_{i\leftarrow} \mid 1 \leq i \leq n\}\end{aligned}$$

Similarly to the set defined above, the $k$ smallest elements from this set are obtained with the following recursive formula:

$$\begin{aligned}\mathcal{M}_{\leftarrow}^{0} &= \varnothing \\ \mathcal{M}_{\leftarrow}^{n} &= \mathcal{M}_{\leftarrow}^{n-1} \cup \left\{\min\ \left(\mathcal{M}_{\leftarrow} \setminus \mathcal{M}_{\leftarrow}^{n-1}\right)\right\}\end{aligned}$$

Depending on the transition times considered, the sum of the $k$ smallest transition *from* activities can be larger or smaller than its counterpart on the sum of the min transitions *to* activities. As both these quantities are lower bounds, the largest value will make the tightest – and therefore most desirable – lower bound:

$$\underline{\mathrm{tt}}(k) = \max \left\{ \begin{array}{l} \sum\limits_{\mathcal{M}_{\rightarrow}^{k-1}} \mathrm{tt}_{i\rightarrow} \\ \sum\limits_{\mathcal{M}_{\leftarrow}^{k-1}} \mathrm{tt}_{i\leftarrow} \end{array} \right.$$

*Example*

We propose here a small example of the lower bounds obtained with the sum of minimal transitions relaxation on a set of transition times between 5 activities. The example matrix $\mathcal{M}$ represents the transitions between activities from this set:

$$\mathcal{M} = \begin{pmatrix} 0 & 2 & 3 & 1 & 4 \\ 7 & 0 & 10 & 12 & 9 \\ 11 & 10 & 0 & 13 & 8 \\ 14 & 10 & 9 & 0 & 12 \\ 15 & 17 & 14 & 11 & 0 \end{pmatrix} \tag{5.6}$$

If we compute the minimal transition by row and by column, we have the following results:

$$\text{Min per Row} = \begin{pmatrix} 1 \\ 7 \\ 8 \\ 9 \\ 11 \end{pmatrix} \qquad \text{Min per Column} = \begin{pmatrix} 7 & 2 & 3 & 1 & 4 \end{pmatrix}$$

And then, by summing the $k$ minimal transitions, the following lower bounds are obtained:

| LB | $k$ | | | | |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 |
| $\sum_{\mathcal{M}^{k-1}_{\rightarrow}} \mathrm{tt}_{i\rightarrow}$ | **0** | **1** | **8** | **16** | **25** |
| $\sum_{\mathcal{M}^{k-1}_{\rightarrow}} \mathrm{tt}_{i\rightarrow}$ | **0** | **1** | 3 | 6 | 10 |
| $\underline{\mathrm{tt}}(k) = \max$ | **0** | **1** | **8** | **16** | **25** |

Table 5.1: Lower bounds obtained by the sum of minimal transitions relaxation for sets of activities of cardinality $n = 1, \ldots, 5$.

### 5.3.2 *Minimum Weight Forest*

Brucker and Thiele [BT96] propose to represent the problem as a graph, where each node corresponds to an activity and directed edges between nodes represent the transition time between corresponding activities. A lower bound for $\underline{\mathrm{tt}}(k)$ is a minimal subset of edges of size $k$ taken from this graph. We propose to strengthen this bound by using Kruskal's algorithm [Kru56] to avoid selecting edges forming a cycle. Even if this relaxation removes cycles, it can lead to a set of transition times where several transitions go to (or leave) the same activity. We stop Kruskal's algorithm as soon as we have collected $k$ edges. The result is a set of edges forming a minimum weight forest (i.e., a set of trees) with exactly $k$ edges.

*Example*

We propose here a small example of the lower bounds obtained with the minimum weight forest relaxation on a set of transition times between 5 activities with transitions defined in Equation (5.6). Applying Kruskal's algorithm [Kru56], we obtain the following lower bounds:

| LB | $k$ | | | | |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 |
| $\underline{\mathrm{tt}}(k)$ | 0 | 1 | 3 | 6 | 10 |

Table 5.2: Lower bounds obtained by the sum of minimal transitions relaxation for sets of activities of cardinality $n = 1, \ldots, 5$.

Graphically, the minimum weight forest obtained for $k = 4$ is shown in Figure 5.1.

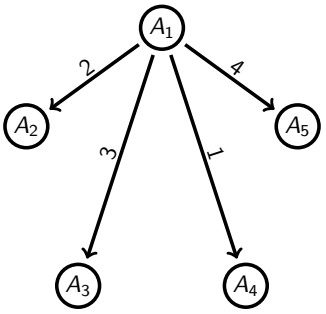



Figure 5.1: Minimum weight forest lower bound for $k = 4$ on example from Equation (5.6).

### 5.3.3 *Dynamic Programming*

Using the graph representation described in Section 5.3.2, we compute a lower bound with a Dynamic Programming approach. The proposed dynamic program allows to compute a shortest walk (not necessarily *simple*) of exactly $k$ edges in the graph. We iteratively compute $\mathrm{DP}(m, u)$ with $1 \leq m \leq k$, that is, the shortest walk of $m$ edges to

node $u$. At each iteration, $\mathrm{DP}(m, u)$ is computed for each node $u$ in the graph using the following update rules:

$$\begin{aligned} \mathrm{DP}(0, u) &= 0 \\ \mathrm{DP}(m+1, u) &= \min_x \left(\mathrm{DP}(m, x) + w_{x,u}\right) \end{aligned}$$

where $x$ is a vertex and $w_{x,u}$ is the weight of the edge from node $x$ to $u$. Once we obtain the value of $\mathrm{DP}(k, u)$ for each node $u$, we can obtain a lower bound as follows:

$$\underline{\mathrm{tt}}(k) = \min_u \ \mathrm{DP}(k, u)$$

This formulation allows the walk of $k$ edges to contain cycles. However, it ensures that a walk (i.e. a single path potentially containing cycles) is computed. A simplified version of the code used in OscaR implementing this Dynamic Programming approach is provided in Appendix A.4.

*Example*

We propose here a small example of the lower bounds obtained with dynamic programming relaxation on a set of transition times between 5 activities with transitions defined in Equation (5.6). With this approach, the $D(m, u)$ values computed iteratively are reported in Table 5.3. From these values, the smallest for each $k$ defines $\underline{\mathrm{tt}}(k)$ as reported in Table 5.4.

| $u$ | $m$ | | | | |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 |
| $A_1$ | 0 | 7 | 9 | 16 | 18 |
| $A_2$ | 0 | 2 | 9 | 11 | 18 |
| $A_3$ | 0 | 3 | 10 | 12 | 19 |
| $A_4$ | 0 | 1 | 8 | 10 | 17 |
| $A_5$ | 0 | 4 | 11 | 13 | 20 |

Table 5.3: Values $D(m, u)$ obtained by the dynamic programming approach.

| LB | $k$ | | | | |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 |
| $\underline{\text{tt}}(k)$ | 0 | 1 | 3 | 6 | 10 |

Table 5.4: Lower bounds obtained by the dynamic programming relaxation for sets of activities of cardinality $n = 1, \ldots, 5$.

### 5.3.4 *Minimum Assignment*

Brucker and Thiele [BT96] propose another lower bound that is computed by considering the problem as a minimum assignment problem. To do so, we duplicate the set of activities, and we try to find a minimum assignment of $k$ edges (i.e., transition times $\text{tt}_{i,j}$) such that each edge has its end points in the two different sets. This problem can be defined by the following linear integer program:

$$
\begin{aligned}
\text{minimize} \quad & \textstyle\sum_{A_i \in T} \sum_{A_j \in T} \text{tt}_{i,j} \cdot x_{i,j} \\
\text{such that} \quad & \textstyle\sum_i \sum_j \; x_{i,j} = k \\
& \textstyle\sum_i \; x_{i,j} \leq 1 \\
& \textstyle\sum_j \; x_{i,j} \leq 1 \\
& x_{i,j} = \{0, 1\}
\end{aligned}
$$

The bound obtained by solving this linear integer program does not contain loops but it does not guarantee that selected edges form a path (i.e., the edges selected may not be successive). Note that this relaxation provides a bound that is at least as tight as the one from obtained by summing the of minimal transitions per line/column. This means that this former relaxation dominates the latter. Nevertheless, as the A simplified version of the code used in OscaR to implement this relaxation is provided in Appendix A.4.

*Example*

We propose here a small example of the lower bounds obtained with the minimum assignment relaxation on a set of transition times between 5 activities with transitions defined in Equation (5.6). Applying this relaxation, we obtain the following lower bounds:

| LB | $k$ | | | | |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 |
| $\underline{\mathrm{tt}}(k)$ | 0 | 1 | 8 | 16 | 25 |

Table 5.5: Lower bounds obtained by the minimum assignment relaxation for sets of activities of cardinality $n = 1, \ldots, 5$.

Graphically, the minimum assignment obtained for $k = 4$ is shown in Figure 5.2.

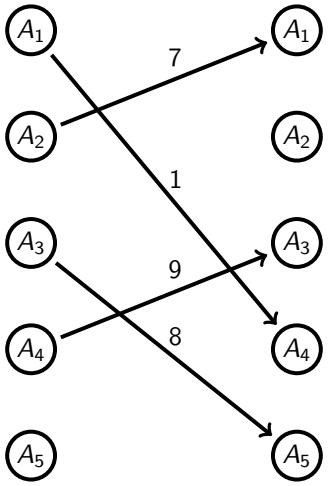



Figure 5.2: Minimum assignment lower bound for $k = 4$ on example from Equation (5.6).

### 5.3.5 *Lagrangian Relaxation*

The exact problem of finding the shortest succession of $k$ transition times can be reduced to finding the shortest-path of length $k$ inside a graph (the graph described in Section 5.3.2). To do so, we add a source node (node 0) and a sink node (node $n + 1$) to this graph such that there is a directed edge of weight 0 from the source to the original

nodes, as well as from the original nodes to the sink. As such, we are able to formulate the following NP-hard problem:

$$
\begin{aligned}
\text{minimize} \quad & \textstyle\sum_i \sum_j \ \mathrm{tt}_{i,j} \cdot x_{i,j} \\
\text{such that} \quad & \textstyle\sum_i \sum_j \ x_{i,j} = k \\
& \textstyle\sum_j \ x_{0,j} - \sum_j \ x_{j,0} = 1 \\
& \textstyle\sum_j \ x_{n+1,j} - \sum_j \ x_{j,n+1} = -1 \\
& \textstyle\sum_j \ x_{i,j} - \sum_j \ x_{j,i} = 0 \\
& x_{i,j} = \{0, 1\}
\end{aligned}
$$

When it is combined to the path constraints, the cardinality constraint $\sum_i \sum_j \ x_{i,j} = k$ makes this problem hard to solve. Therefore, we compute a lower bound by solving a Lagrangian relaxation of the above linear integer program. It uses the Bellman-Ford algorithm [Moo59, Bel58] to compute the shortest path of length $k$ and if a negative cycle is detected, a classic Linear Integer Programming approach is used.

*Example*

We propose here a small example of the lower bounds obtained with the Lagrangian relaxation on a set of transition times between 5 activities with transitions defined in Equation (5.6). Applying this relaxation, we obtain the following lower bounds:

| LB | $k$ | | | | |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 |
| $\underline{\mathrm{tt}}(k)$ | 0 | 1 | 5 | 10 | 16 |

Table 5.6: Lower bounds obtained by the Lagrangian relaxation for sets of activities of cardinality $n = 1, \ldots, 5$.

### 5.3.6 *Exact TSP*

Instead of using relaxations that might provide loose values, one could desire to compute the exact value of $\underline{\mathrm{tt}}(k)$ for all $k \in [1, n]$. In [VB02], a

Dynamic Programming approach is introduced to compute the smallest sequence of transitions of a set of activities with the first activity fixed. This approach iteratively computes the smallest sequence of transitions for *all the possible* subsets of activities. Furthermore, for each subset, all the activities from the subset are considered once being the first of the sequence. Formally, this Dynamic Programming approach uses the following recurrence rules:

$$\begin{aligned} \mathrm{tt}_{A_i \to \{A_i\}} &= 0 \\ \mathrm{tt}_{A_i \to \mathcal{A}' \cup \{A_i\}} &= \min_{A_j \in \mathcal{A}'} \left\{ \mathrm{tt}_{ij} + \mathrm{tt}_{A_j \to \mathcal{A}'} \right\} \end{aligned}$$

where $\mathcal{A}$ is the set of all activities, $\mathcal{A}' \subseteq \mathcal{A}$ and $\mathrm{tt}_{A_j \to \mathcal{A}'}$ is the smallest sequence of transitions for activity subset $\mathcal{A}'$ such that $A_j$ is the first of the sequence. Once all these values have been obtained, the values of $\underline{\mathrm{tt}}(k)$ for all $k \in [1, n]$ are easily obtained. For each $k$, all the values $\mathrm{tt}_{A_j \to \mathcal{A}'}$ where $|\mathcal{A}'| = k$ are collected and the minimal value is retained. Formally, this is done as follows:

$$\underline{\mathrm{tt}}(k) = \min_{|\mathcal{A}'|=k} \min_{A_j \in \mathcal{A}'} \mathrm{tt}_{A_j \to \mathcal{A}'}$$

The computation of all these values grows dramatically with the number of activities considered. For a set of $n$ activities, the time complexity of this approach is $\mathcal{O}(n^2 \cdot 2^n)$. Furthermore, all these values also have to be stored in memory and the space complexity is $\mathcal{O}(n \cdot 2^n)$. This method is not adapted for large matrices and does not scale. Hence, we will not use it to compute $\underline{\mathrm{tt}}(k)$. We only report this approach for comparison reasons.

*Example*

We propose here a small example of the lower bounds obtained with the exact TSP on a set of transition times between 5 activities with transitions defined in Equation (5.6). Applying this method, we obtain the following lower bounds:

| LB | $k$ | | | | |
|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 |
| $\underline{\mathrm{tt}}(k)$ | 0 | 1 | 8 | 17 | 25 |

Table 5.7: Lower bounds obtained by the exact TSP procedure for sets of activities of cardinality $n = 1, \ldots, 5$.

### 5.3.7 *Comparison of Lower Bounds*

We propose here an example of the lower bounds obtained on a set of transition times between 15 activities. The example matrix $\mathcal{M}$ represents the transitions between activities from this set:

$$\mathcal{M} = \begin{pmatrix}
0 & 13 & 17 & 11 & 10 & 17 & 9 & 13 & 19 & 13 & 15 & 18 & 10 & 5 & 5 \\
13 & 0 & 18 & 5 & 10 & 12 & 7 & 10 & 6 & 13 & 14 & 12 & 11 & 12 & 15 \\
9 & 17 & 0 & 19 & 11 & 7 & 17 & 17 & 16 & 17 & 12 & 12 & 11 & 14 & 14 \\
16 & 12 & 13 & 0 & 5 & 15 & 7 & 12 & 15 & 15 & 12 & 7 & 6 & 7 & 12 \\
12 & 12 & 16 & 8 & 0 & 12 & 10 & 15 & 18 & 13 & 7 & 15 & 9 & 11 & 15 \\
15 & 13 & 9 & 12 & 6 & 0 & 10 & 10 & 12 & 12 & 5 & 14 & 6 & 15 & 12 \\
15 & 8 & 14 & 13 & 13 & 11 & 0 & 5 & 14 & 8 & 12 & 20 & 13 & 8 & 17 \\
13 & 16 & 14 & 18 & 12 & 6 & 12 & 0 & 9 & 11 & 7 & 16 & 12 & 16 & 12 \\
18 & 16 & 22 & 14 & 6 & 18 & 16 & 14 & 0 & 11 & 13 & 13 & 15 & 16 & 11 \\
12 & 5 & 15 & 10 & 6 & 17 & 12 & 15 & 11 & 0 & 13 & 17 & 15 & 17 & 17 \\
10 & 12 & 9 & 17 & 13 & 13 & 5 & 10 & 15 & 7 & 0 & 9 & 7 & 10 & 8 \\
9 & 12 & 15 & 10 & 14 & 8 & 6 & 8 & 16 & 13 & 6 & 0 & 13 & 14 & 5 \\
16 & 14 & 7 & 13 & 17 & 14 & 6 & 11 & 17 & 14 & 18 & 15 & 0 & 14 & 6 \\
17 & 9 & 12 & 13 & 5 & 17 & 8 & 13 & 15 & 8 & 10 & 18 & 5 & 0 & 11 \\
10 & 23 & 15 & 17 & 20 & 19 & 19 & 16 & 16 & 18 & 17 & 15 & 15 & 15 & 0
\end{pmatrix}$$

We computed the lower bounds of the sum of the $k$ transitions taking place in any set of $n = k + 1$ activities. The lower bounds were obtained with the relaxations mentioned earlier. In Table 5.8, we report the time (in milliseconds) taken by each of these relaxations to compute the lower bounds of transition times in a set from size 1 to 15 (i.e., containing 0 to 14 transitions). The results seem to show that the min assignment relaxation is time consuming with respect to other relaxations. However, to illustrate the time needed on larger instance, Table 5.9 reports the time (in milliseconds) taken by the relaxations to compute the lower bounds on a set of $n = 50$ activities. The time taken by a Lagrangian relaxation as described earlier greatly increases.

As expected, the time needed for summing the min elements per line, Kruskal's minimum spanning tree and dynamic programming relaxations seem to follow a quadratic increase. The time needed to compute the exact value, with the exact TSP procedure described earlier is relatively small for $n = 15$. However, for $n = 50$, it would be too large to be reported here (or even to be computed someday).

| LB | Time (ms) |
|---|---|
| Sum of Min. Trans. | 0.08 |
| Min. Weight Forest | 0.26 |
| Dyn. Prog. | 0.08 |
| Min. Ass. | 33.00 |
| Lag. Relax. | 534.40 |
| Exact TSP | 67.86 |

Table 5.8: Time taken by the relaxations to compute the lower bounds of a set of transitions based on the cardinality of the set ($n = 15$).

| LB | Time (ms) |
|---|---|
| Sum of Min. Trans. | 0.2 |
| Min. Weight Forest | 2.9 |
| Dyn. Prog. | 1.0 |
| Min. Ass. | 1,628.8 |
| Lag. Relax | 55,924.4 |
| Exact TSP | $+\infty$ |

Table 5.9: Time taken by the relaxations to compute the lower bounds of a set of transitions based on the cardinality of the set ($n = 50$).

The lower bounds obtained by the various relaxations on cardinalities $k = 0, \ldots, 14$ (i.e., sets of 1 to 15 activities) are reported in Table 5.10. As expected, the lower bound obtained by the sum of the minimal elements by line or column is alway inferior or equal to the lower bounds obtained by the min assignment. No relaxation proposes a lower bound that is above the ones obtained by other relaxations for

all the possible cardinalities. It is therefore useful to keep several relaxations such that the best possible lower bound can be used. This is even more useful when considering that these relaxations will only be used in a pre-computation step. The line $\underline{\text{tt}}(k) = \max$ takes the maximal lower bounds obtained by the relaxations for every $k$. The last line, *Exact TSP* reports the exact value obtained when solving all the TSPs of size $k$ with the procedure defined earlier. Again, this approach is totally impractical since there are $2^n$ TSPs to solve to obtain these values (one TSP for each possible subset of the $n$ activities). The lower bounds obtained are very tight, they are below by a maximum of three units of the exact TSP value for some cardinalities. Therefore, these relaxations seem to provide tight lower bounds and have the advantage to scale in terms of time (and memory) needed to compute them.

| LB | $k$ | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 |
| Sum of Min. Trans. | **0** | **5** | **10** | **15** | 20 | 25 | 30 | 35 | 40 | 45 | 51 | 57 | 64 | 71 | 78 |
| Min. Weight Forest | **0** | **5** | **10** | **15** | 20 | 25 | 30 | 35 | 40 | 45 | 50 | 55 | 61 | 67 | 74 |
| Dyn. Prog. | **0** | **5** | **10** | **15** | **21** | **26** | **31** | **36** | **42** | **47** | 52 | 57 | 63 | 68 | 73 |
| Min. Ass. | **0** | **5** | **10** | **15** | 20 | 25 | 30 | 35 | 40 | 45 | 51 | 58 | **66** | **74** | **83** |
| Lag. Relax. | **0** | 4 | 9 | **15** | 20 | 25 | 30 | 35 | 41 | **47** | **53** | **59** | **66** | 72 | 78 |
| $\underline{\text{tt}}(k) = \max$ | **0** | **5** | **10** | **15** | **21** | **26** | **31** | **36** | **42** | **47** | **53** | **59** | **66** | **74** | **83** |
| Exact TSP | 0 | 5 | 10 | 15 | 21 | 27 | 32 | 38 | 44 | 49 | 54 | 61 | 69 | 76 | 85 |

Table 5.10: Lower bounds obtained by the relaxations for sets of activities of cardinality $n = 1, \ldots, 15$.

## 5.4 THE THETA-TREE STRUCTURE

As introduced in [Vilo4b], the $\mathcal{O}(n \log n)$ propagation algorithms for unary resource use the so-called $\Theta$-tree data structure. We propose to extend it in order to integrate transition times while keeping the same time complexities for all its operations.

A $\Theta$-tree is a balanced binary tree in which each leaf represents an activity from a set $\Theta$ and internal nodes gather information about the set of (leaf) activities under this node. For an internal node $v$, we denote by Leaves($v$), the leaf activities under $v$. Leaves are ordered

in non-decreasing order of the est of the activities. That is, for two activities $A_i$ and $A_j$, if $\text{est}_i < \text{est}_j$, then $A_i$ is represented by a leaf node that is on the left of the leaf node representing $A_j$. This ensures the following property :

$$\forall A_i \in \text{Left}(v), \forall A_j \in \text{Right}(v) : \text{est}_i \leq \text{est}_j$$

where left$(v)$ and right$(v)$ are respectively the left and right children of $v$, and Left$(v)$ and Right$(v)$ denote respectively Leaves(left$(v)$) and Leaves(right$(v)$).

A node $v$ contains pre-computed values about Leaves$(v)$: $\Sigma\text{d}_v$ represents the sum of the durations of activities in Leaves$(v)$ and $\text{ect}_v$ is the ect of Leaves$(v)$. More formally, the values maintained in an internal node $v$ are defined as follows:

$$\Sigma\text{d}_v = \sum_{A_j \in \text{Leaves}(v)} d_j$$

$$\text{ect}_v = \text{ect}_{\text{Leaves}(v)} = \max_{\Theta' \subseteq \text{Leaves}(v)} (\text{est}_{\Theta'} + d_{\Theta'})$$

For a given leaf $l$ representing an activity $i$, the values of $\Sigma\text{d}_l$ and $\text{ect}_l$ are respectively $d_i$ and $\text{ect}_i$. In [Vilo7] Vilím has shown that for a node $v$ these values only depend on the values defined in its left$(v)$ and right$(v)$ children. The incremental update rules introduced in [Vilo7] are:

$$\Sigma\text{d}_v = \Sigma\text{d}_{\text{left}(v)} + \Sigma\text{d}_{\text{right}(v)}$$

$$\text{ect}_v = \max \begin{cases} \text{ect}_{\text{right}(v)} \\ \text{ect}_{\text{left}(v)} + \Sigma\text{d}_{\text{right}(v)} \end{cases}$$

An example of a classic Θ-tree is given in Figure 5.3.

When transition times are considered, the $\text{ect}_v$ values computed in the internal nodes of the Θ-tree may only be a loose lower-bound since it is only based on the earliest start times and durations of activities. We strengthen the estimation of the earliest completion times (written $\text{ect}^*$) by also considering transition times. We add another value inside

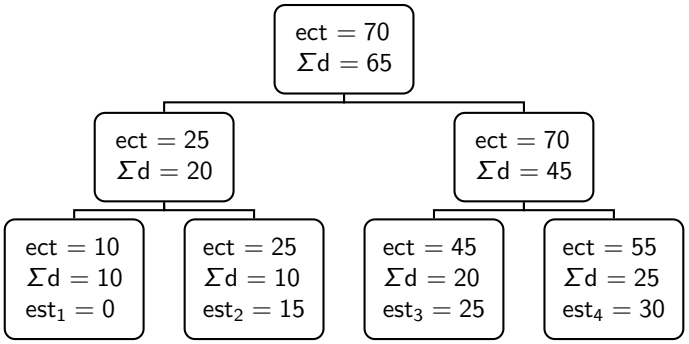



Figure 5.3: Classic Θ-tree as described in [Vilo7].

the nodes: $n_v$ is the cardinality of Leaves($v$): $n_v = |\text{Leaves}(v)|$. The new update rules for the nodes of a Θ-tree are:

$$\Sigma d_v = \Sigma d_{\text{left}(v)} + \Sigma d_{\text{right}(v)}$$

$$n_v = n_{\text{left}(v)} + n_{\text{right}(v)}$$

$$ect^*_v = \max \begin{cases} ect^*_{\text{right}(v)} \\ ect^*_{\text{left}(v)} + \Sigma d_{\text{right}(v)} + \underline{tt}(n_{\text{right}(v)}) \end{cases}$$

In these update rules, the term $\underline{tt}(n_{\text{right}(v)})$ is surprising since it supposes that we consider right($v$) transitions for a set of right($v$) activities, where normally one should consider only right($v$) − 1 transitions. The additional transition is included to represent the transition between the activities in the left subtree and those in the right subtree.

As an example, let us consider the set of four activities used in the Θ-tree example of Figure 5.3. Let us assume that the associated transition times are as defined in the matrix $\mathcal{M}$ of Figure 5.4. The lower bounds for sets of activities of different cardinality are reported next to the matrix. With the new update rules defined earlier, we obtain the extended Θ-tree presented in Figure 5.5. Note that the values of ect* in the internal nodes are larger than the values of ect reported in the classic Θ-tree (Figure 5.3).

**Theorem 1.** *For any set of activities for which a lower bound of its ect is computed with an extended Θ-tree, the ect* computed in any node satisfies the following property:*

$$ect_v \leq ect^*_v \leq ect^{\diamond}_{Leaves(v)} = \max_{\Theta' \subseteq Leaves(v)} \left( est_{\Theta'} + d_{\Theta'} + \underline{tt}(|\Theta'|) \right)$$

$$\mathcal{M} = \begin{pmatrix} 0 & 10 & 13 & 18 \\ 12 & 0 & 15 & 15 \\ 10 & 18 & 0 & 20 \\ 19 & 11 & 16 & 0 \end{pmatrix}$$

| LB | $k$ | | | |
|---|---|---|---|---|
| | 0 | 1 | 2 | 3 |
| Sum of Min. Trans. | 0 | 10 | 20 | 33 |
| Min. Weight Forest | 0 | 10 | 20 | 31 |
| Dyn. Prog. | 0 | 10 | 20 | 32 |
| Min. Ass. | 0 | 10 | 20 | 33 |
| Lag. Relax. | 0 | 10 | 20 | 32 |
| $\underline{\mathrm{tt}}(k) = \max$ | 0 | 10 | 20 | 33 |

Figure 5.4: Example of transition time matrix and associated lower bounds of transition times occurring in a set of activities.

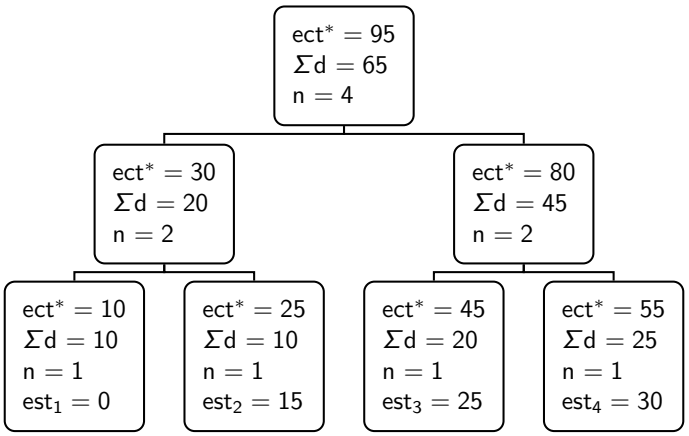



Figure 5.5: Extended Θ-tree for transition times. The ect* values reported in the internal nodes have been computed using the update rule of the extended Θ-tree.

*Proof.* The proof is similar to the proof of Proposition 7 in [Vil07]. The only term added to the original ect update in a classic Θ-tree is the

$\underline{\text{tt}}()$ term. However, for any set of activities $\Theta$, the following inequality holds:

$$\underline{\text{tt}}(|\Theta|) \geq \underline{\text{tt}}(|\Theta \cap \text{Left}(v)|) + \underline{\text{tt}}(|\Theta \cap \text{Right}(v)|)$$

This inequality is itself a direct consequence of the fact that $\underline{\text{tt}}(k)$ is monotonic in $k$. □

Since the new update rules are also executed in constant time for one node, we keep the time complexities of the initial $\Theta$-tree structure from [Vil07]. Inserting or removing an activity inside the tree requires to update all its parents up to the root. Considering a set of $n$ activities, the $\Theta$-tree contains at most $n$ leaves. As theta trees are balanced binary trees, leaves have $\mathcal{O}(\log(n))$ parents. Hence, inserting or removing a single activity is performed in $\mathcal{O}(\log(n))$. Therefore, the insertion or removal of all activities in the tree has a time complexity of $\mathcal{O}(n \log(n))$. The complexities of all the actions performed on an extended $\Theta$-tree are reported in Table 5.11.

| Operation | | Time Complexity |
|---|---|---|
| Clear | $\Theta \leftarrow \varnothing$ | $\mathcal{O}(n \log(n))$ |
| Remove | $\Theta \leftarrow \Theta \cup \{A_i\}$ | $\mathcal{O}(\log(n))$ |
| Add | $\Theta \leftarrow \Theta \setminus \{A_i\}$ | $\mathcal{O}(\log(n))$ |
| Get | $\text{ect}^*_\Theta$ | $\mathcal{O}(1)$ |
| Fill | $\Theta \leftarrow T$ | $\mathcal{O}(n \log(n))$ |

Table 5.11: Worst-case time complexities of operations on extended $\Theta$-tree.

*Extending the $\Theta$-$\Lambda$-tree with Transition Times*

The Edge Finding (EF) algorithm requires an extension of the original $\Theta$-tree, called $\Theta$-$\Lambda$-tree [Vil07]. This extension is used to obtain a time efficient EF algorithm. In this extension, in addition to the activities included in a $\Theta$-tree, leaves can be marked as *gray nodes*. Gray nodes represent activities that are not anymore in the set $\Theta$. However, they allow to easily compute $\text{ect}_\Theta$ if *a single one* of the gray activities were included in $\Theta$. Let us consider the set of gray activities $\Lambda$ such that

$\Lambda \cap \Theta = \varnothing$. Our interest lies in the computation of the largest ect obtained by including *only one* activity from $\Lambda$ into $\Theta$:

$$\overline{\text{ect}}_{(\Theta,\Lambda)} = \max_{A_i \in \Lambda} \text{ect}_{\Theta \cup \{A_i\}}$$

In addition to $\Sigma \text{d}_v$ and $\text{ect}^*_v$, the $\Theta$-$\Lambda$-tree structure also maintains $\overline{\Sigma \text{d}}_v$ and $\overline{\text{ect}}^*_v$, respectively corresponding to $\Sigma \text{d}_v$ and $\text{ect}^*_v$, *if* a single gray activity in the sub-tree rooted in $v$ maximizing $\text{ect}_v$ *was* included:

$$\overline{\text{ect}}^*_{(\Theta,\Lambda)} = \max_{A_i \in \Lambda} \text{ect}^*_{\Theta \cup \{A_i\}}$$

The update rule for $\overline{\Sigma \text{d}}_v$ remains the same as the one described in [Vil07]. However, following a similar reasoning as the one used for the extended $\Theta$-tree, we add the $\overline{\text{n}}_v$ value. The update rule for $\overline{\text{ect}}^*_v$ is also modified in order to take into account the transition times inside $\text{Leaves}(v)$. The updated rules for the extended $\Theta$-$\Lambda$-tree are the following:

$$\overline{\Sigma \text{d}}_v = \max \begin{cases} \overline{\Sigma \text{d}}_{\text{left}(v)} + \Sigma \text{d}_{\text{right}(v)} \\ \Sigma \text{d}_{\text{left}(v)} + \overline{\Sigma \text{d}}_{\text{right}(v)} \end{cases}$$

$$\overline{\text{ect}}^*_v = \max \begin{cases} \overline{\text{ect}}^*_{\text{right}(v)} \\ \overline{\text{ect}}^*_{\text{left}(v)} + \Sigma \text{d}_{\text{right}(v)} + \underline{\text{tt}}(\text{n}_{\text{right}(v)}) \\ \text{ect}^*_{\text{left}(v)} + \overline{\Sigma \text{d}}_{right(v)} + \underline{\text{tt}}(\overline{\text{n}}_{\text{right}(v)}) \end{cases}$$

$$\overline{\text{n}}_v = \begin{cases} \text{n}_v + 1 & \text{if the subtree rooted in } v \text{ contains a gray node} \\ \text{n}_v & \text{otherwise} \end{cases}$$

This extended $\Theta$-$\Lambda$-tree allows us to efficiently observe how the $\text{ect}^*$ of a set of activities is impacted if a single activity is added to this set. This information allows the EF algorithm to perform propagation efficiently; finding the "responsible" activity $\arg\max_{A_i} \text{ect}_{\Theta \cup \{A_i\}}$ (required by EF) is done similarly to [Vil07]. An example of $\Theta$-$\Lambda$-tree based on the example from Figure 5.4 and Figure 5.5 is displayed in Figure 5.6.

Similarly to the reasoning applied for the extended $\Theta$-tree, the time complexities remain the same as the ones from the original $\Theta$-$\Lambda$-tree structure from [Vil07]. The time complexities of the various operations performed on a $\Theta$-$\Lambda$-tree are recalled in Table 5.12.

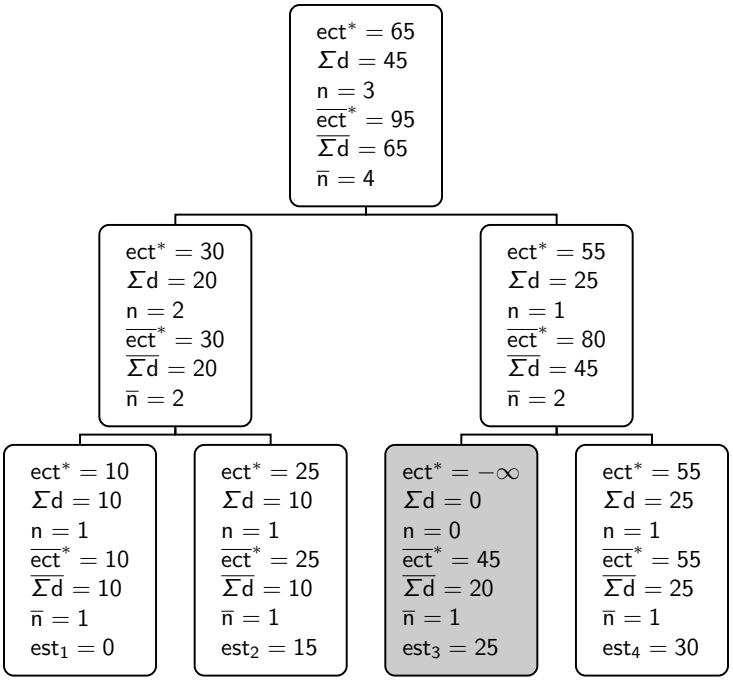



Figure 5.6: Extended Θ-Λ-tree for transition times with one gray activity. The $\text{ect}^*$ and $\overline{\text{ect}}^*$ values reported in the internal nodes have been computed using the update rules of the extended Θ-Λ-tree.

| Operation | | Time Complexity |
|---|---|---|
| Clear | $(\Theta, \Lambda) \leftarrow (\varnothing, \varnothing)$ | $\mathcal{O}(1)$ |
| Fill | $(\Theta, \Lambda) \leftarrow (T, \varnothing)$ | $\mathcal{O}(n \log(n))$ |
| Gray | $(\Theta, \Lambda) \leftarrow (\Theta \backslash \{A_i\}, \Lambda \cup \{A_i\})$ | $\mathcal{O}(\log(n))$ |
| Insert | $\Theta \leftarrow \Theta \cup \{A_i\}$ | $\mathcal{O}(\log(n))$ |
| Remove | $\Lambda \leftarrow \Lambda \backslash \{A_i\}$ | $\mathcal{O}(\log(n))$ |
| $\overline{\text{Get}}$ | $\overline{\text{ect}}^*_{(\Theta,\Lambda)}$ | $\mathcal{O}(1)$ |
| Get | $\text{ect}^*_\Theta$ | $\mathcal{O}(1)$ |

Table 5.12: Worst-case time complexities of operations on extended Θ-Λ-tree.

## 5.5 PROPAGATION ALGORITHMS

In [Vil07], a propagation procedure for the unary resource constraint is defined. This propagation procedure consists of a propagation loop

including Overload Checking (OC), Detectable Precedences (DP), Not-First/Not-Last (NF/NL) and Edge Finding (EF) algorithms. The first three rely on the $\Theta$-tree while the latter employs the $\Theta$-$\Lambda$-tree. Some small modifications can be done to these algorithms to obtain an efficient propagation procedure making use of knowledge about transition times.

The four mentioned propagation algorithms use a $\Theta$-tree or a $\Theta$-$\Lambda$-tree to compute $\text{ect}_\Theta$ on a set of activities $\Theta$. OC checks if $\text{ect}_\Theta > \text{lct}_\Theta$. DP, NF/NL and EF rely on a set of rules that potentially allow to update the est or lct of an activity. They all incrementally add/remove activities to a set of activities $\Theta$ while maintaining the value $\text{ect}_\Theta$. When a rule is triggered by the consideration of a given activity, the est or lct of this activity can be updated according to the current value of $\text{ect}_\Theta$. In the rest of this section, we describe the extended versions of the four propagation algorithms mentioned earlier.

Each of these algorithms potentially updates only one of the bounds of an activity: either its est or its lct. However, it is easy to make these algorithms potentially update both bounds with the help of a single transformation. To each activity $A_i$ can be associated a *mirror activity* $A_i^m$. This mirror activity defines two *views* on the start and end variables from the original activity and shares its duration variable. Given a propagator $p$, a view [SSo8] is represented by two functions $\phi$ and $\phi^{-1}$ that are composed with $p$ to obtain the desired propagator $\phi \circ p \circ \phi^{-1}$. The $\phi$ function transforms the input domain and $\phi^{-1}$ applies the inverse transformation to the propagator's output domain. The mirror activity $A_i^m$ has the following views on the variables of $A_i$:

$$s_i^m = -e_i$$
$$e_i^m = -s_i$$
$$d_i^m = d_i$$

To update both bounds of activities, algorithms are run twice: once on the original activities and once on the mirror activities.

### 5.5.1 *Overload Checking*

The Overload Checking (OC) rule [WSo4] applies on any subset of activities using the same unary resource. Let us consider an arbitrary set of activities $\Omega \subseteq T$ where $T$ is the set of all activities using a given

unary resource. The OC rule states that if activities from the set $\Omega$ cannot be processed within the bounds of $\Omega$, then no solution exists (i.e., a failure has been detected). Formally, for any set of activities $\Omega \subseteq T$, a failure is detected when the following OC rule is satisfied:

$$\text{est}_\Omega + d_\Omega > \text{lct}_\Omega$$

This OC rule can be extended to include transition times between activities. In addition to the sum of durations of activities from $\Omega$, the transitions occurring in the set can be considered. Formally, for any set of activities $\Omega \subseteq T$, a failure is detected when the following extended OC rule is satisfied:

$$\text{ect}_\Omega > \text{lct}_\Omega$$

As this extended OC rule can be applied for any $\Omega$, thanks to Equation (5.5), it is possible to substitute $\text{ect}_\Omega$ by $\text{ect}^*_\Omega$. Therefore, the extended OC rule can be simplified as follows:

$$\text{ect}^*_\Omega > \text{lct}_\Omega$$

As the exact value for $\text{ect}_\Omega$ is hard to compute in practice, it can be substituted with a lower bound. The lower bound $\text{ect}^*_\Omega$ obtained with the extended $\Theta$-tree data structure introduced in Section 5.4 can be used for the extended OC rule.

The extended Overload Checking algorithm from Algorithm 5.5.1 applies the extended OC rule on various sets $\Omega \subseteq T$. The sets $\Omega$ are built by adding activities one by one in non-decreasing order of lct (line 3). For each of the sets obtained, the extended OC rule is tested in line 4. If the extended OC rule is true for a set $\Omega$, then a failure has been detected and the algorithm returns **Failure**. If no set $\Omega$ has triggered the extended OC rule, no failure has been detected; the algorithm ends and returns **Suspend**.

The worst time complexity of Algorithm 5.5.1 is $\mathcal{O}(n \log(n))$ where $n = |T|$. Indeed, it starts with an empty $\Theta$-tree and it performs $n$ insertions in the tree, each insertion being performed in $\mathcal{O}(\log(n))$. It is important to mention that the extended OC algorithm from Algorithm 5.5.1 is only a *checker*. This means that it only checks the satisfiability of the current domains from variables in its scope, but it does not modify the domains themselves.

*Example*

Let us consider a small example of four activities using a unary resource and subject to transition times. The activities are represented

**Algorithm 5.5.1 :** Overload Checking

```
1 Θ ← ∅
2 for A_i in non-decreasing order of lct_i do
3 |   Θ ← Θ ∪ {A_i}
4 |   if ect*_Θ > lct_i then
5 |   |   return Failure
6 |   end
7 end
8 return Suspend
```

in Figure 5.7; the transition times matrix and the lower bound of transitions in a set of activities by cardinality are reported in Figure 5.8.

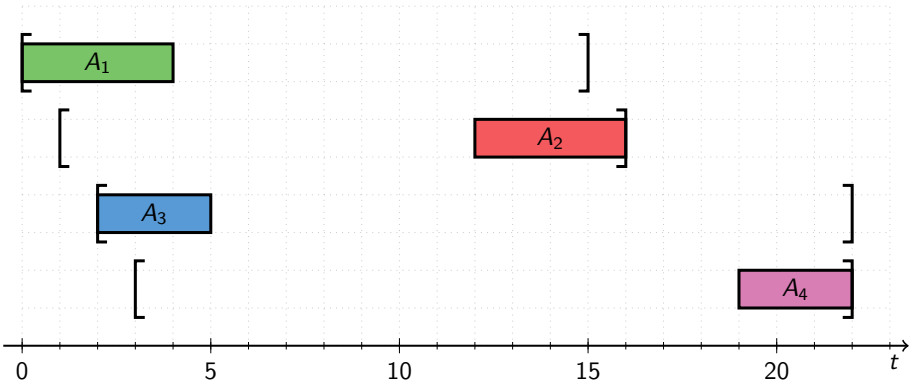



Figure 5.7: Example of activities for which the Overload Checking algorithm detects a failure.

The OC algorithm adds activities in a Θ-tree in non-decreasing order of $\text{lct}_i$. After every insertion, $\text{ect}_\Theta$ is compared to the $\text{lct}_i$ of the last activity $A_i$ added; if $\text{ect}_\Theta > \text{lct}_i$ then a failure occurs. In the case of the example from Figures 5.7 and 5.8, after adding the four activities $A_1, A_2, A_3$ and $A_4$ in the Θ-tree, its content is as depicted in Figure 5.9. The last activity added was $A_4$ and we have $\text{ect}_\Theta > \text{lct}_4$. Therefore, the OC rule has detected a failure.

$$\mathcal{M} = \begin{pmatrix} 0 & 5 & 5 & 4 \\ 4 & 0 & 3 & 4 \\ 5 & 6 & 0 & 7 \\ 5 & 7 & 5 & 0 \end{pmatrix}$$

| LB | $k$ | | | |
|---|---|---|---|---|
| | 0 | 1 | 2 | 3 |
| Sum of Min. Trans. | **0** | **3** | 7 | **12** |
| Min. Weight Forest | **0** | **3** | 7 | 11 |
| Dyn. Prog. | **0** | **3** | **8** | **12** |
| Min. Ass. | **0** | **3** | 7 | **12** |
| Lag. Relax. | **0** | **3** | 7 | 11 |
| $\underline{\mathrm{tt}}(k) = \max$ | 0 | 3 | 8 | 12 |
| Exact TSP | 0 | 3 | 8 | 12 |

Figure 5.8: Example of transition times and lower bound of transitions within a set of activities by cardinality.

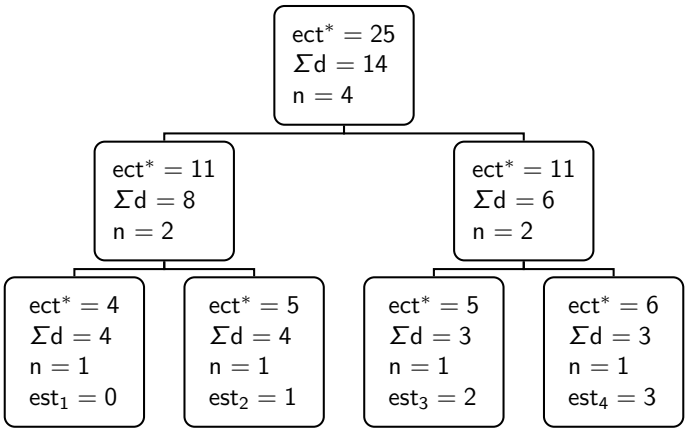



Figure 5.9: Θ-tree obtained after the iterative addition of the activities from Figure 5.7.

### 5.5.2 *Detectable Precedence*

The Detectable Precedence rule [Vil04a] updates the est of an activity with respect to the set of activities preceding it. When an activity $A_j$ precedes another activity $A_i$, $A_j \ll A_i$, the following deduction can be performed:

$$\mathrm{est}_i \geq \mathrm{ect}_j$$

Considering a third activity $A_k$ also preceding $A_i$, $A_k \ll A_i$, the same deduction can be applied. However, the update obtained with single precedences is relatively weak and does not bring important bound change. if we consider that both $A_j$ and $A_k$ precedes $A_i$, $\{A_j, A_k\} \ll A_i$, then a stronger deduction is possible. Indeed, with $\Omega = \{A_j, A_k\}$, then the following deduction can be made:

$$\text{est}_i \geq \text{ect}_\Omega$$

For a given activity $A_i$, this reasoning can be extended for all the activities preceding it. As specified in [Vil07], the set of all *detectable preceding activities* for an activity $A_i$ is referred to as $\text{DPrec}(A_i)$ and is defined as follows:

$$\text{DPrec}(A_i) = \{A_j \mid j \in T \ \wedge \ \text{lst}_j < \text{ect}_i \ \wedge \ j \neq i\}$$

The detectable precedence rule updates $\text{est}_i$ according to $\text{ect}_{\text{DPrec}(A_i)}$, its set of detectable preceding activities:

$$\text{est}_i = \max \begin{cases} \text{est}_i \\ \\ \text{ect}_{\text{DPrec}(A_i)} \end{cases}$$

To include transition times in the DP update rule, the $\text{ect}_{\text{DPrec}(A_i)}$ can be replaced by $\text{ect}^*_{\text{DPrec}(A_i)}$. However, in addition to the transitions in the set $\text{DPrec}(A_i)$ (that are embedded in $\text{ect}^*_{\text{DPrec}(A_i)}$), there will be an additional transition from the last activity of $\text{DPrec}(A_i)$ to $A_i$. As it is not possible to know what activity in $\text{DPrec}(A_i)$ will come last, one could count the minimal transition from any of the activities $A_j \in \text{DPrec}(A_i)$ to $A_i$. Computing the minimal transition from a set to an activity has a time complexity in $\mathcal{O}(n)$. This could be pre-computed for every activity $A_i$ and every possible set of activities $\Omega \subseteq T \setminus \{A_i\}$. Unfortunately, there exists $2^{n-1}$ such set for every activity $A_i$, leading to $n \cdot 2^{n-1}$ values to pre-compute. Regardless of the time needed to compute all these values, the memory needed to store them could not scale. Therefore, we propose to use again a lower bound for this transition that is the smallest transition from any activity $A_j \in T \setminus \{A_i\}$ to $A_i$, $\text{tt}_{i\leftarrow}$:

$$\text{tt}_{i\leftarrow} = \min_{j \neq i} \ \text{tt}_{j,i}$$

This means that only $n$ values have to be pre-computed and each of these values is obtained in $\mathcal{O}(n)$. With the addition of this minimal

transition, the extended detectable precedences update rule is defined as follows:

$$\text{est}_i = \max \begin{cases} \text{est}_i \\ \text{ect}^*_{\text{DPrec}(A_i)} + \text{tt}_{i\leftarrow} \end{cases}$$

The extended Detectable Precedences algorithm depicted in Algorithm 5.5.2 applies the extended DP rule to all activities on a resource. It goes through all activities $A_i$ in non-decreasing order of $\text{ect}_i$ (loop starting at line 4) and adds all activities that are in $\text{DPrec}(A_i)$ in a $\Theta$-tree (lines 5-7). Then, at line 9, the new bound for $\text{est}_i$ is registered in a cache $\text{est}'_i$. The new bounds are registered in a cache and not directly applied to domains in order to keep the $\Theta$-tree consistent. Finally, once all activities have been browsed, the new bounds kept in cache are applied to the domains in line 11. This is done in Algorithm 5.5.3 that applies the bounds in cache to the domains and stops if a failure has been detected.

**Algorithm 5.5.2 :** Detectable Precedence

```
1  Θ ← ∅
2  Q ← Queue of A_j in non-decreasing order of lst_j
3  A_j ← Pop(Q)
4  for A_i in non-decreasing order of ect_i do
5  |  while ect*_i > lst_j do
6  |  |  Θ ← Θ ∪ {A_j}
7  |  |  A_j ← Pop(Q)
8  |  end
9  |  est'_i ← max { est_i
   |               { ect*_{Θ\{A_i}} + tt_{i←}
10 end
11 return UpdateESTs()
```

The worst time complexity of Algorithm 5.5.2 is $\mathcal{O}(n \log(n))$ where $n = |T|$. Indeed, it starts with an empty $\Theta$-tree and it performs $n$ insertions in the tree, each insertion being performed in $\mathcal{O}(\log(n))$. It also performs $\mathcal{O}(n)$ successive removals and insertions (both performed in $\mathcal{O}(\log(n))$) when getting the value $\text{ect}^*_{\Theta\setminus\{A_i\}}$. Indeed, to compute $\text{ect}^*_{\Theta\setminus\{A_i\}}$, one first removes $A_i$ from the tree, then get the desired value, then reinsert $A_i$ back in the tree to restore its original state.

**Algorithm 5.5.3 :** Update ESTs

```
1 forall the A_i do
2 |   est_i ← est'_i
3 |   if a failure has occurred then
4 |   |   return Failure
5 |   end
6 end
7 return Suspend
```

*Example*

Let us consider a small example of three activities using a unary resource and subject to transition times. The activities are represented in Figure 5.10; the transition times matrix and the lower bound of transitions in a set of activities by cardinality are reported in Figure 5.11.

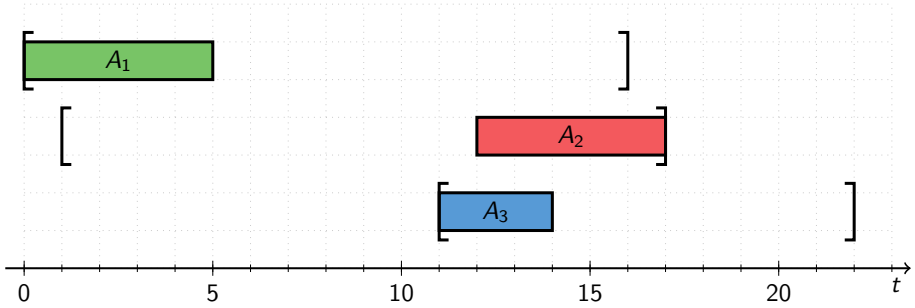



Figure 5.10: Example of activities for which the Detectable Precedences algorithm updates the bounds of an activity.

When the DP algorithm considers the activity $A_3$, both $A_1$ and $A_2$ are added in the $\Theta$-tree that is shown in Figure 5.12. The update performed for $A_3$ is the following one:

$$\text{est}'_3 = 17 = \max \begin{cases} \text{est}_3 = 11 \\ \text{ect}^*_{\Theta \setminus \{A_3\}} + \text{tt}_{3\leftarrow} = 12 + 5 = 17 \end{cases}$$

Once this update has been applied on the domain of $A_3$, we obtain the domains as shown in Figure 5.13.

$$\mathcal{M} = \begin{pmatrix} 0 & 4 & 6 \\ 2 & 0 & 5 \\ 4 & 3 & 0 \end{pmatrix}$$

| LB | $k$ | | |
|---|---|---|---|
| | 0 | 1 | 2 |
| Sum of Min. Trans. | **0** | **2** | **5** |
| Min. Weight Forest | **0** | **2** | **5** |
| Dyn. Prog. | **0** | **2** | **5** |
| Min. Ass. | **0** | **2** | **5** |
| Lag. Relax. | **0** | **2** | 4 |
| $\underline{\mathrm{tt}}(k) = \max$ | 0 | 2 | 5 |
| Exact TSP | 0 | 2 | 5 |

Figure 5.11: Example of transition times and lower bound of transitions within a set of activities by cardinality.

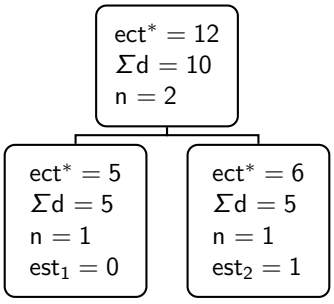



Figure 5.12: Θ-tree obtained after the addition of $A_1$ and $A_2$ from Figure 5.10.

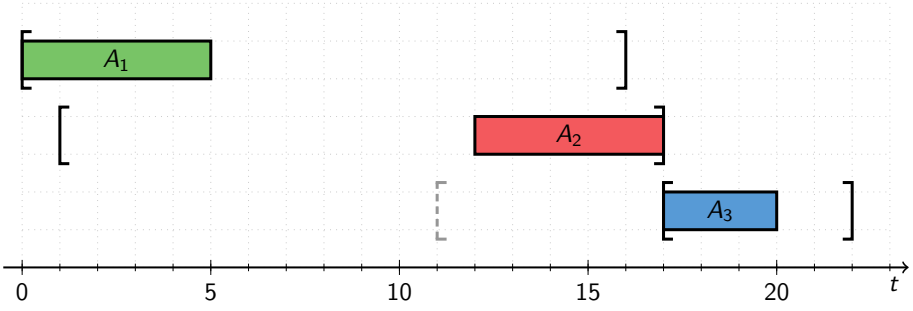



Figure 5.13: Activities from Figure 5.10 after the Detectable Precedences algorithm from Algorithm 5.5.2 updates $\mathrm{est}_3$. The dashed gray bracket is the former value of $\mathrm{est}_3$.

### 5.5.3 *Not-First/Not-Last*

The Not-First and Not-Last rules [Vil04a] are two symmetric update rules. They respectively update the est and lct of an activity. We only present here the Not-Last rule and algorithm; the Not-First rule is obtained easily with the mirror transformation we have mentioned earlier.

The Not-Last reasoning is applied on an activity $A_i$ and a set of activities $\Omega \subseteq T$ such that $A_i \notin \Omega$. If the activity $A_i$ cannot be scheduled after the set of activities $\Omega$ (i.e., $A_i$ cannot be the last activity in $\Omega \cup \{A_i\}$), then $\text{lct}_i$ can be updated. An activity $A_i$ cannot be scheduled after a set of activities $\Omega$ if:

$$\text{ect}_\Omega > \text{lst}_i$$

In this case, there is at least one activity from $\Omega$ that has to be scheduled before $A_i$. Therefore, the following update can be performed:

$$\text{lct}_i = \min \begin{cases} \text{lct}_i \\ \max\limits_{A_j \in \Omega} \text{lst}_j \end{cases}$$

It is easy to extend the Not-Last rule such that they take transition times into account. The extended rule to detect if an activity $A_i$ cannot be scheduled after a set of activities $\Omega$ is obtained by substituting $\text{ect}^*_\Omega$ to $\text{ect}_\Omega$:

$$\text{ect}^*_\Omega > \text{lst}_i$$

If such case is detected, we know that there should be at least one activity $A_j \in \Omega$ scheduled before $A_i$. We also know that there will be a transition time between $A_i$ and the last activity of $\Omega$. Following a similar reasoning as the one proposed for DP, the last activity of $\Omega$ is not known and therefore the transition to take into account is also unknown. We thus propose to use as lower bound the smallest transition from $A_i$ to any activity $A_j \in T$:

$$\text{tt}_{i\rightarrow} = \min_{j \neq i} \text{tt}_{i,j}$$

Hence, when an activity $A_i$ cannot be scheduled after a set $\Omega$, we can perform the following extended update:

$$\text{lct}_i = \min \begin{cases} \text{lct}_i \\ \max\limits_{A_j \in \Omega} \text{lst}_j - \text{tt}_{i\rightarrow} \end{cases}$$

For a given activity $A_i$, any set $\Omega$ that could apply the NL update rule should be a subset of $\text{NLSet}(A_i)$:

$$\text{NLSet}(A_i) = \{A_j \mid j \in T \ \wedge \ \text{lst}_j < \text{lct}_i \ \wedge \ j \neq i\}$$

For any set of activities $\Omega \subseteq \text{NLSet}(A_i)$, we have the following property:

$$\max_{A_j \in \Omega} \text{lst}_j \leq \max_{A_k \in \text{NLSet}(A_i)} \text{lst}_k$$

Therefore, it is not mandatory to consider the exact $\text{NLSet}(A_i)$; considering only a subset of it can be enough to update $\text{lct}_i$.

The Not-Last algorithm depicted in Algorithm 5.5.4 applies the extended NL rule to all activities applying on a resource. It goes through all activities $A_i$ in non-decreasing order of $\text{lct}_i$ (loop starting at line 4). For each $A_i$, the set $\text{NLSet}(A_i)$ is iteratively built in the loop from line 5 to line 6 by adding all potential activities in a $\Theta$-tree. The activities included in $\Theta$ might contain $A_i$ itself; therefore, when checking if the NL update can be applied, we have to consider the set $\Theta \setminus \{A_i\}$. In line 9, we check if $A_i$ has to be scheduled before at least one activity from $\Theta \setminus \{A_i\}$. In case the NL rule can be applied, the new bound for $\text{lct}_i$ is registered in a cache $\text{lct}'_i$ at line 10. Similarly to the DP algorithm, the cache is used to ensure values in the $\Theta$-tree are still consistent. Finally, once all activities have been browsed, the new bounds kept in cache are applied to the domains in line 13. This is done in Algorithm 5.5.3 that applies the bounds in cache to the domains and stops if a failure has been detected.

The worst time complexity of Algorithm 5.5.4 is $\mathcal{O}(n \log(n))$ where $n = |T|$. Similarly to Algorithm 5.5.2, it starts with an empty $\Theta$-tree and it performs $n$ insertions in the tree, each insertion being performed in $\mathcal{O}(\log(n))$. It also performs $\mathcal{O}(n)$ successive removals and insertions (both performed in $\mathcal{O}(\log(n))$) when getting the value $\text{ect}^*_{\Theta \setminus \{A_i\}}$.

*Example*

Let us consider a small example of three activities using a unary resource and subject to transition times. The activities are represented in Figure 5.14; the transition times matrix and the lower bound of transitions in a set of activities by cardinality are reported in Figure 5.15.

The NL algorithm from Algorithm 5.5.4 begins its main loop with $A_3$ (since it has the smallest lct). The activities $A_1$, $A_2$ and $A_2$ are successively added in the $\Theta$-tree in the loop from line 5-6. Then, the Not-Last

**Algorithm 5.5.4 :** Not-Last

1 $\Theta \leftarrow \varnothing$
2 $Q \leftarrow$ Queue of $A_j$ in non-decreasing order of $\text{lst}_j$
3 $A_j \leftarrow$ `Peek(`$Q$`)`
4 **for** $A_i$ in non-decreasing order of $\text{lct}_i$ **do**
5  **while** $\text{lct}_i > \text{lst}_j$ **do**
6  $\Theta \leftarrow \Theta \cup \{A_j\}$
7  $A_j \leftarrow$ `Pop(`$Q$`)`
8  **end**
9  **if** $\text{ect}_{\Theta \setminus \{A_i\}} > \text{lst}_i$ **then**
10  $\text{lct}'_i \leftarrow \min \begin{cases} \text{lct}_i \\ \text{lst}_j - \text{tt}_{i\rightarrow} \end{cases}$
11  **end**
12 **end**
13 **return** `UpdateLCTs()`

**Algorithm 5.5.5 :** Update LCTs

1 **forall the** $A_i$ **do**
2  $\text{lct}_i \leftarrow \text{lct}'_i$
3  **if** a failure has occurred **then**
4  **return** Failure
5  **end**
6 **end**
7 **return** Suspend

rule is checked in line 9. This rule checks if $\text{ect}^*_{\Theta \setminus \{A_3\}} > \text{lst}_3$. At this moment, $\Theta \setminus \{A_3\} = \{A_1, A_2\}$ and the computation of $\text{ect}^*_{\Theta \setminus \{A_3\}}$ is shown in Figure 5.16. As the NL rule should apply, the update performed for $A_3$ is the following one:

$$\text{lct}'_3 = 12 = \min \begin{cases} \text{lct}_3 = 15 \\ \text{lst}_2 - \text{tt}_{3\rightarrow} = 14 - 2 = 12 \end{cases}$$

Once this update has been applied on the domain of $A_3$, we obtain the domains as shown in Figure 5.17.

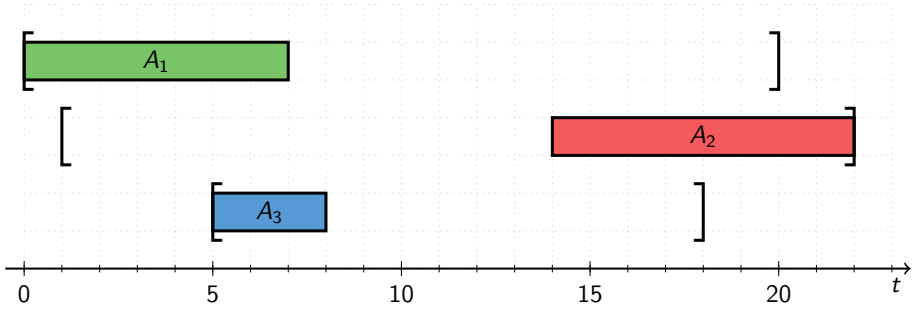



Figure 5.14: Example of activities for which the Not-Last algorithm updates the bounds of an activity.

$$\mathcal{M} = \begin{pmatrix} 0 & 4 & 2 \\ 6 & 0 & 5 \\ 4 & 2 & 0 \end{pmatrix}$$

| LB | $k$ | | |
|---|---|---|---|
| | 0 | 1 | 2 |
| Sum of Min. Trans. | **0** | **2** | **4** |
| Min. Weight Forest | **0** | **2** | **4** |
| Dyn. Prog. | **0** | **2** | **4** |
| Min. Ass. | **0** | **2** | **4** |
| Lag. Relax. | **0** | 1 | **4** |
| $\underline{\mathrm{tt}}(k) = \max$ | 0 | 2 | 4 |
| Exact TSP | 0 | 2 | 4 |

Figure 5.15: Example of transition times and lower bound of transitions within a set of activities by cardinality.

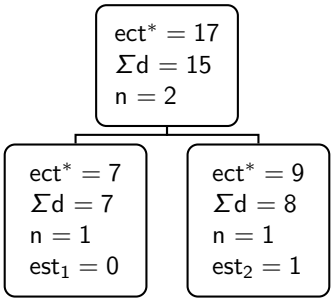



Figure 5.16: $\Theta$-tree obtained after the addition of $A_1$ and $A_2$ from Figure 5.14.

### 5.5.4 *Edge Finding*

The Edge Finding rule [BL96] updates the est of an activity with respect to a set of activities. Let us consider a set of activities $\Omega \subseteq T$ and

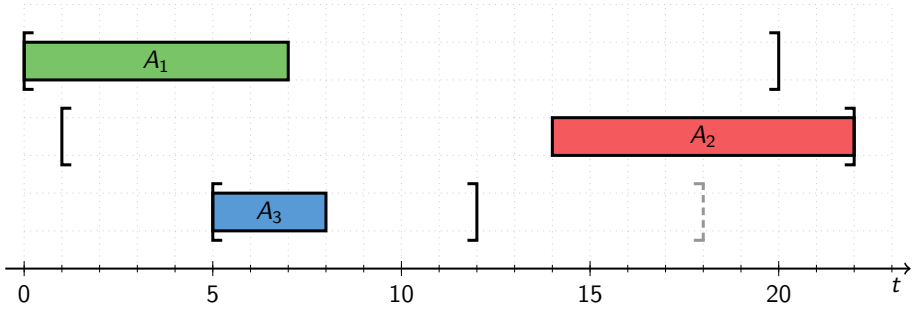



Figure 5.17: Activities from Figure 5.14 after the Not-Last algorithm from Algorithm 5.5.4 updates $\text{lct}_3$. The dashed gray bracket is the former value of $\text{lct}_3$.

an activity $A_i \notin \Omega$. If the latest completion time of the set $\Omega$ is smaller than the earliest completion time of the set $\Omega$ including $A_i$, then $A_i$ has to be scheduled after all activities from the set $\Omega$. Formally, for any set $\Omega \subseteq T$ and any activity $A_i \notin \Omega$, the EF rule detects that $A_i$ has to be scheduled after all other activities in $\Omega$ if:

$$\text{ect}_{\Omega \cup \{A_i\}} > \text{lct}_\Omega$$

If this rule holds, then $A_i$ has to be scheduled after $\Omega$ and the following update rule can be applied:

$$\text{est}_i = \max \begin{cases} \text{est}_i \\ \text{ect}_\Omega \end{cases}$$

One can easily include transition times in the EF rule. The extended rule to detect if an activity $A_i$ must be scheduled after a set of activities $\Omega$ is obtained by substituting $\text{ect}^*_\Omega$ to $\text{ect}_\Omega$:

$$\text{ect}^*_{\Omega \cup \{A_i\}} > \text{lct}_\Omega$$

If such case is detected, we know that activities from $\Omega$ should be scheduled before $A_i$. We also know that there will be a transition time between the last activity of $\Omega$ and $A_i$. Following a similar reasoning as the one proposed for DP and NL, the last activity of $\Omega$ is not known and therefore the transition to take into account is also unknown. We thus propose to use as lower bound the smallest transition from any activity $A_j \in T$ to $A_i$:

$$\text{tt}_{i\leftarrow} = \min_{j \neq i} \text{tt}_{j,i}$$

Hence, when an activity $A_i$ must be scheduled after a set $\Omega$, we can perform the following extended update:

$$\text{est}_i = \max \begin{cases} \text{est}_i \\ \text{ect}^*_\Omega + \text{tt}_{i\leftarrow} \end{cases}$$

To ease the computation of $\text{ect}^*_{\Omega\cup\{A_i\}}$, we can take advantage of the $\Theta$-$\Lambda$-tree structure. Indeed, the $\Theta$-$\Lambda$-tree structure allows to answer the question *What would be the ect if one gray activity is added to $\Omega$?*. Therefore, the value $\overline{\text{ect}}^*_{(\Omega,\Lambda)}$ is the value of $\text{ect}_{\Omega\cup\{A_i\}}$ where $A_i \in \Lambda$ is the gray activity that will increase the most the ect. The extended EF rule can therefore be substituted with the following:

$$\overline{\text{ect}}^*_{(\Omega,\Lambda)} > \text{lct}_\Omega$$

If this rule is verified, then the est of the gray activity in the $\Theta$-$\Lambda$-tree responsible for $\overline{\text{ect}}^*_{(\Omega,\Lambda)}$ can be updated with the extended EF update rule described earlier.

The extended Edge Finding algorithm depicted in Algorithm 5.5.6 applies the extended EF rule to all activities executed on a unary resource. In line 1, a $\Theta$-$\Lambda$-tree is initialized with all the activities set in $\Theta$ and none in $\Lambda$ (i.e., no gray activity yet). It goes through all activities $A_j$ in non-decreasing order of $\text{lct}_i$ (loop starting at line 4). At each iteration, the current activity $A_j$ is grayed in line 8; then $A_j$ is set to be the next activity. At line 10, the extended EF rule is checked. If the EF rule applies, the gray activity responsible for $\overline{\text{ect}}^*_{(\Omega,\Lambda)}$ is identified in line 11; its est is updated with the extended EF update rule in line 12 and it is finally removed at line 13. The loop starting in line 10 can be applied on several gray activities while the EF rule holds. In line 5, the OC rule is checked. This is not mandatory if the OC algorithm is included in the fixed point but it can speed up the propagation by detecting a failure earlier.

The worst time complexity of Algorithm 5.5.6 is $\mathcal{O}(n\log(n))$ where $n = |T|$. It starts with a full $\Theta$-$\Lambda$-tree and it grays $n$ activities; graying an activity is performed in $\mathcal{O}(\log(n))$. It also removes at most $n$ activities from the set of gray activities (each being also performed in $\mathcal{O}(\log(n))$).

*Example*

Let us consider a small example of four activities using a unary resource and subject to transition times. The activities are represented

**Algorithm 5.5.6 :** Edge Finding

```
1  (Θ, Λ) ← (Ω, ∅)
2  Q ← Queue of A_j in non-increasing order of lct_j
3  A_j ← Pop(Q)
4  while Size(Q) > 1 do
5  |  if ect*_Θ > lct_j then
6  |  |  return Failure
7  |  end
8  |  (Θ, Λ) ← (Θ \ {A_j}, Λ ∪ {A_j})
9  |  A_j ← Pop(Q)
10 |  while ect*_(Θ,Λ) > lct_j do
11 |  |  A_i ← gray activity responsible for ect*_(Θ,Λ)
12 |  |  est'_i ← max{est_i, ect_Θ}
13 |  |  Λ ← Λ \ A_i
14 |  end
15 end
16 return UpdateESTs()
```

in Figure 5.18; the transition times matrix and the lower bounds of transitions in a set of activities by cardinality are reported in Figure 5.19.

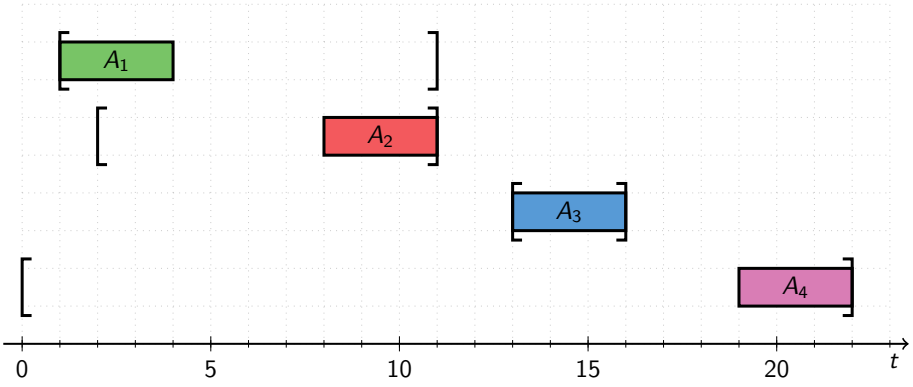



Figure 5.18: Example of activities for which the Not-Last algorithm updates the bounds of an activity.

The EF algorithm from Algorithm 5.5.6 first fills the Θ-Λ-tree in line 1 such that $(\Theta, \Lambda) = (\{A_1, A_2, A_3, A_4\}, \varnothing)$. Then, the loop starting in line 4 begins with $A_j = A_4$. In line 5, the OC rule is checked, but

$$\mathcal{M} = \begin{pmatrix} 0 & 3 & 4 & 3 \\ 6 & 0 & 2 & 5 \\ 4 & 7 & 0 & 2 \\ 4 & 7 & 8 & 0 \end{pmatrix}$$

| LB | $k$ | | | |
|---|---|---|---|---|
| | 0 | 1 | 2 | 3 |
| Sum of Min. Trans. | **0** | **2** | **4** | **7** |
| Min. Weight Forest | **0** | **2** | **4** | **7** |
| Dyn. Prog. | **0** | **2** | **4** | **7** |
| Min. Ass. | **0** | **2** | **4** | **7** |
| Lag. Relax. | **0** | 1 | **4** | 6 |
| $\underline{\text{tt}}(k) = \max$ | 0 | 2 | 4 | 7 |
| Exact TSP | 0 | 2 | 4 | 7 |

Figure 5.19: Example of transition times and lower bound of transitions within a set of activities by cardinality.

it does not apply here; no failure has been detected. The activity $A_4$ is grayed in line 8, leading to $(\Theta, \Lambda) = (\{A_1, A_2, A_3\}, \{A_4\})$. The activity $A_j$ is set to $A_3$ in line 9. As the EF rule holds: $\overline{\text{ect}}^*_{(\Theta,Lambda)} > \text{lct}_3$, we enter the loop starting in line 10. The loop begins by retrieving the gray activity responsible for the value of $\overline{\text{ect}}^*_{(\Theta,Lambda)}$ (line 11); in this case, the responsible gray activity is $A_4$. The EF update rule is applied in line 12 as follows:

$$\text{est}'_4 = 18 = \max \begin{cases} \text{est}_4 = 0 \\ \text{ect}^*_\Theta + \text{tt}_{4\leftarrow} = 16 + 2 = 18 \end{cases}$$

Once this update has been applied on the domain of $A_4$, we obtain the domains as shown in Figure 5.21.

### 5.5.5 *Unary Resource with Transition Times Fixed Point*

The four algorithms described earlier for the unary resource with transition times constraint will be called in a fixed point algorithm described in Chapter 1. As all these propagators implement the same constraint, we propose to integrate them in their own propagation loop, as proposed in [Vil07] for the classic unary resource constraint. Indeed, as these algorithms are not idempotent and as they propose a different

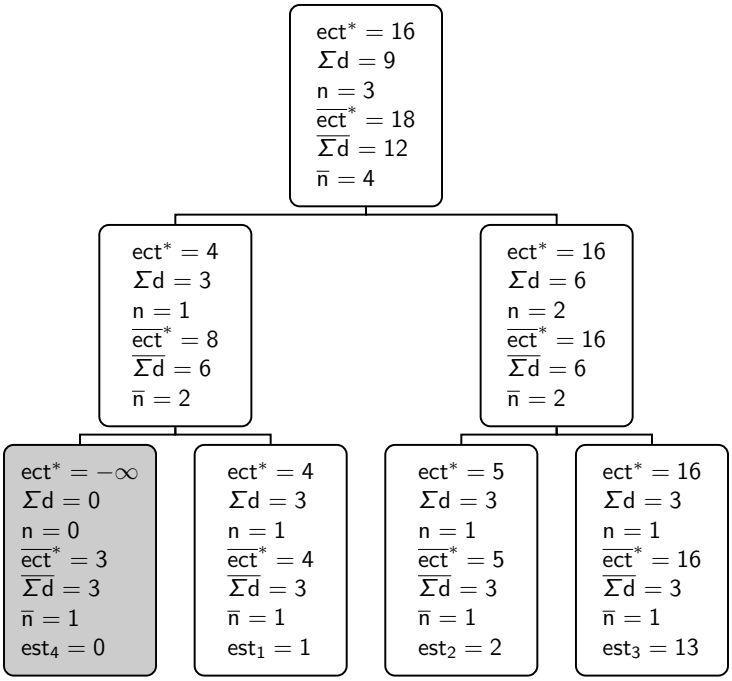



Figure 5.20: $\Theta$-$\Lambda$-tree obtained when $(\Theta, \Lambda) = (\{A_1, A_2, A_3\}, \{A_4\})$ with activities from Figure 5.18.

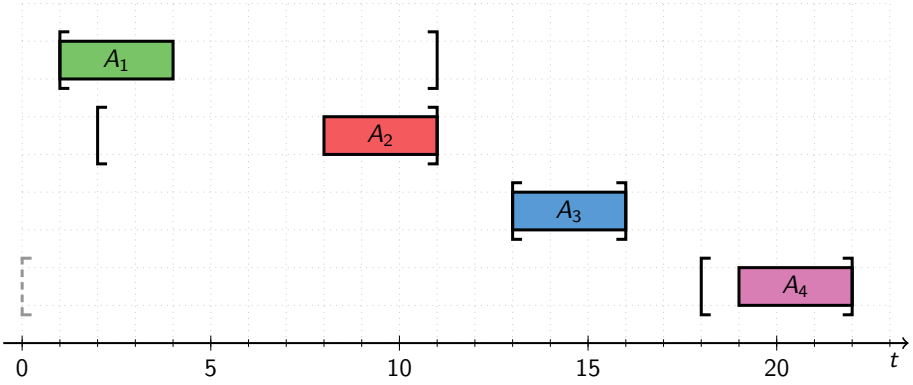



Figure 5.21: Activities from Figure 5.18 after the Edge Finding algorithm from Algorithm 5.5.6 updates $\text{est}_4$. The dashed gray bracket is the former value of $\text{est}_4$.

pruning, they will trigger events that will require them to be run again until no further pruning can be achieved.

As the algorithms described earlier rely on several lower bounds, the pruning achieved is not sufficient to ensure the correctness of the

constraint; i.e., the pruning achieved by these four algorithms could accept an unfeasible assignment as a solution. To ensure that the fixed point of the constraint is correct, we propose to add $n^2$ binary decomposition equations that ensure the correctness of the propagation loop as proposed in Equation (5.1) and recalled here:

$$\forall i, j : (e_i + \mathrm{tt}_{i,j} \leq s_j) \vee (e_j + \mathrm{tt}_{j,i} \leq s_i)$$

Propagating each single equation is done in constant time $\mathcal{O}(1)$. When a single variable has been updated, only $n$ equations have to be propagated. Hence, the propagation of a single variable update is performed in $\mathcal{O}(n)$. The propagation of all variables implies to propagate all the equations and is achieved in $\mathcal{O}(n^2)$. Most of the time only a few variables have been updated and trigger propagation of the equations in which they are implied. Therefore, this propagation is performed in amortized linear time.

As mentioned in [RVW06], the order in which the algorithms will be called inside this propagation loop can have a non-negligible impact on the performance. Furthermore, we must consider whether or not the pruning of each propagator is worth it; the gain in terms of propagation of one algorithm might not compensate the time required to make it run. To make a complete study, one should consider all the possible orderings of the five propagators (OC, DP, NFNL, EF and binary decomposition) including or not the four *relaxation* propagation algorithms (OC, DP, NFNL, EF). It should compare all permutations of the 5 propagators plus all permutations of 4 propagators while not including one of the four *relaxation* propagation algorithms and so on. We have the following possible number of propagation loops to consider:

$$5! + 4 * 4! + 6 * 3! + 4 * 2! + 1 = 120 + 96 + 36 + 8 + 1 = 261$$

Looking for the best possible ordering would require to compare the time taken by propagation of these 261 possible orderings on a large set of instances. Lacking the time and resources to do so, we haven't performed this study and this remains an open question for further work.

According to some small experimental results, we have come up with an ordering that performs well on a wide range of various instances. This ordering is reported in Figure 5.22. The Overload Checking algorithm was not included in the loop since its reasoning is included in the Edge Finding algorithm. This propagation loop includes

all the reasoning and pruning rules of the four propagation algorithms described before (and also the binary decomposition). This propagation loop performs an *internal fixed point*. Therefore, this propagation loop can be seen as a single *block* that will be added on the propagation queue when one of the events to which it is registered is triggered. In our case, this propagation loop registers to any event implying a domain change on the start, end or duration variables in the scope of the constraint it implements.

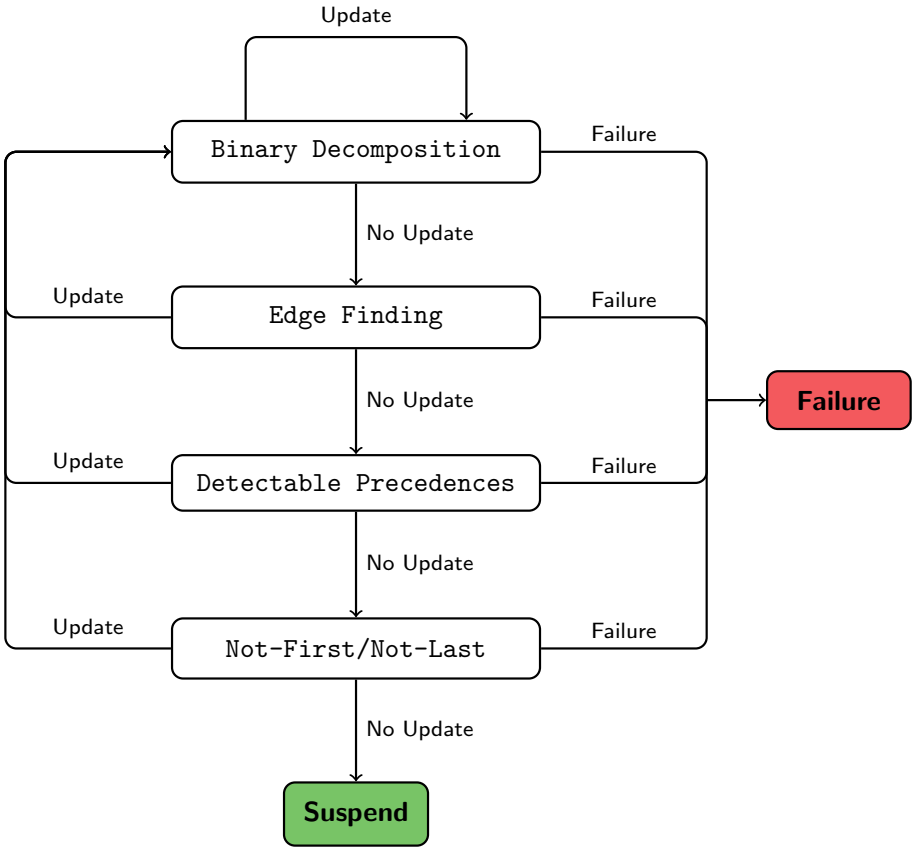



Figure 5.22: Propagation loop for the unary resource with transition times constraint.

## 5.6 EVALUATION

To evaluate our propagation procedure, we used the OscaR solver [Osc12] and ran instances on AMD Opteron processors (2.7 GHz). For

each considered instance, we used the three following filtering procedures for the unary resource with transition times constraint:

BINARY DECOMPOSITION
Binary decomposition constraints ($\phi_b$) given in Equation (5.1). For efficiency reasons, a global dedicated propagator has been implemented instead of posting reified constraints.

UNARY RESOURCE + BINARY DECOMPOSITION
Binary decomposition constraints given in Equation (5.1) with the unary resource global constraint from Vilím [Vil07] ($\phi_{b+u}$).

UNARY RESOURCE WITH TRANSITION TIMES
The constraint introduced in this thesis ($\phi_{uTT}$). The propagation is performed using the internal propagation loop described in Figure 5.22.

### 5.6.1 *Considered benchmarks*

We have constructed instances considering transition times from famous Job Shop benchmarks. For a given benchmark $\mathcal{B}$, in each instance, we added generated transition times between activities, while ensuring that triangular inequality always holds. From $\mathcal{B}$, we generated new benchmarks $\mathcal{B}_{(a,b)}$ inside which the instances are expanded by transition times uniformly picked between $a$% and $b$% of $\overline{D}$, where $\overline{D}$ is the average duration of all activities in the original instance.

We generated instances from the well-known Taillard's instances (available at `http://mistic.heig-vd.ch/taillard/problemes.dir/ordonnancement.dir/ordonnancement.html`). From each Taillard's instance, we generated two instances for a given pair $(a, b)$, where the following pairs have been used : $(50, 100)$, $(50, 150)$, $(50, 200)$, $(100, 150)$, $(100, 200)$ and $(150, 200)$. This has allowed us to generate 960 new instances (available at `http://becool.info.ucl.ac.be/resources/benchmarks-unary-resource-transition-times`).

### 5.6.2 *Comparison of the Three Models*

In order to present fair results regarding the benefits that are only provided by our propagators, we first followed the methodology introduced in [CLS15]. An overview of this methodology is provided in Chapter 3. Additionally, we have made measurements using a static

search strategy, as it cannot be influenced by the additional pruning provided by our propagation procedure.

*Potential of the constraint*

In brief, the approach presented in [CLS15] proposes to pre-compute a search tree using the filtering that prunes the less – the *baseline* propagator – and then to *replay* this search tree using the different studied filtering procedures. The point is to only measure the time gain provided by the propagation, by decoupling the gain provided by the search strategy (while still being able to use dynamic search strategies) from the one provided by the propagation. We used $\phi_b$ as the baseline filtering, and the *Conflict Ordering Search* (COS) [GHPS15] strategy to construct the search tree, as this strategy has been shown to achieve good performances in scheduling. The search tree construction time was limited to 600 seconds. We then constructed *performance profiles* as described in [CLS15]. Performance profiles in the context of the comparison of propagators are described in Chapter 3.

*Performance Profiles on Number of Backtracks*

The performance profiles for backtracks are illustrated in Figure 5.23. We can see that the pruning of $\phi_{uTT}$ is really important. For more than 35% of the instances, the pruning of $\phi_{uTT}$ is at least ten times larger than the one achieved by $\phi_b$. And it is at least twice as large than the one proposed by $\phi_b$ for about 88% of the instances.

On the other hand, the pruning achieved by $\phi_{b+u}$ is larger than the one of $\phi_b$ on only 65% of the instances. The pruning of $\phi_{b+u}$ is at least twice as large as the one of $\phi_b$ on only 2% of the instances. We can deduce from this that using classic unary resource propagators when considering problems involving transition times is inefficient. Indeed, as we will see with performance profiles for resolution time, the classic unary resource propagators will only add computational overhead to binary decomposition pruning while not being able to reduce the search space in most cases.

*Performance Profiles on Resolution Time*

The performance profiles for replay time of sequences of nodes generated for 600 seconds with $\phi_b$ and the COS heuristics are illustrated in Figure 5.24. For at least 45% of the instances, $\phi_{uTT}$ is at least twice

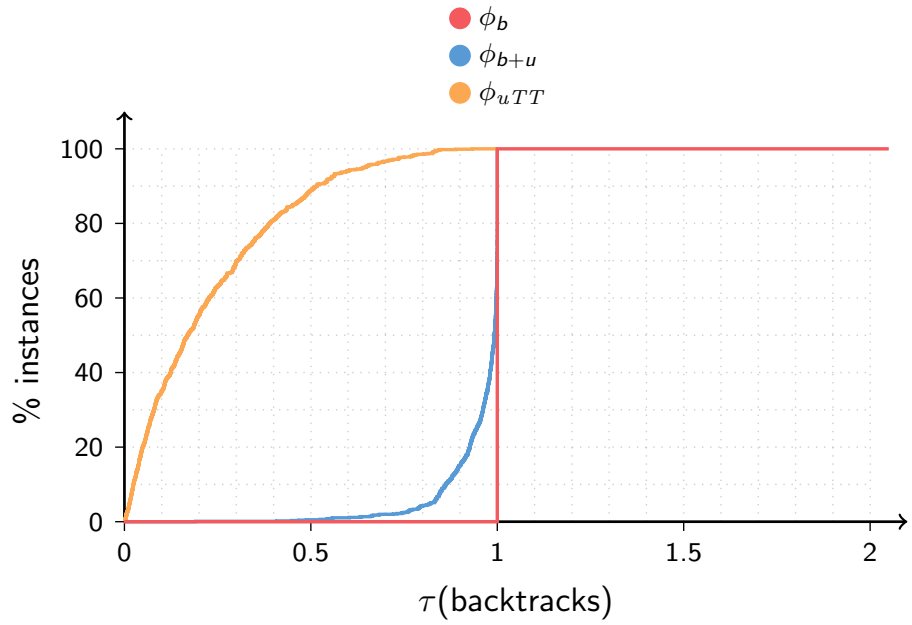



Figure 5.23: Performance profiles of the three models for the number of backtracks.

faster than $\phi_b$. Furthermore, $\phi_{uTT}$ is at least as fast as $\phi_b$ on 68% of the instances. However, $\phi_b$ is faster than $\phi_{uTT}$ for about 32% of the instances. The same performance profiles are represented up to the maximal performance ratio obtained in Figure 5.25. In this latter representation, one can observe that $\phi_{uTT}$ is at most 7.5 times slower than $\phi_b$. On the other hand, $\phi_{b+u}$ is faster than $\phi_b$ on only at least 3% of the instances. It has poor time performances, it is at worst 7.5 times slower than $\phi_b$ on 60% of the instances. This confirms the conclusion of the backtrack performance profile from Figure 5.23 stating that using the binary decomposition and the classic unary resource propagators is inefficient for problems considering transition times.

*Evaluation over a Static Search Strategy*

The replay evaluation strategy with performance profiles used earlier has shown the potential of our propagator in comparison with a classic binary decomposition. However, the replay evaluation greatly favors the base model, in our case the binary decomposition. Indeed, the branching strategy chosen will favor the exploration of the search space for the base model, during the construction of the sequence of

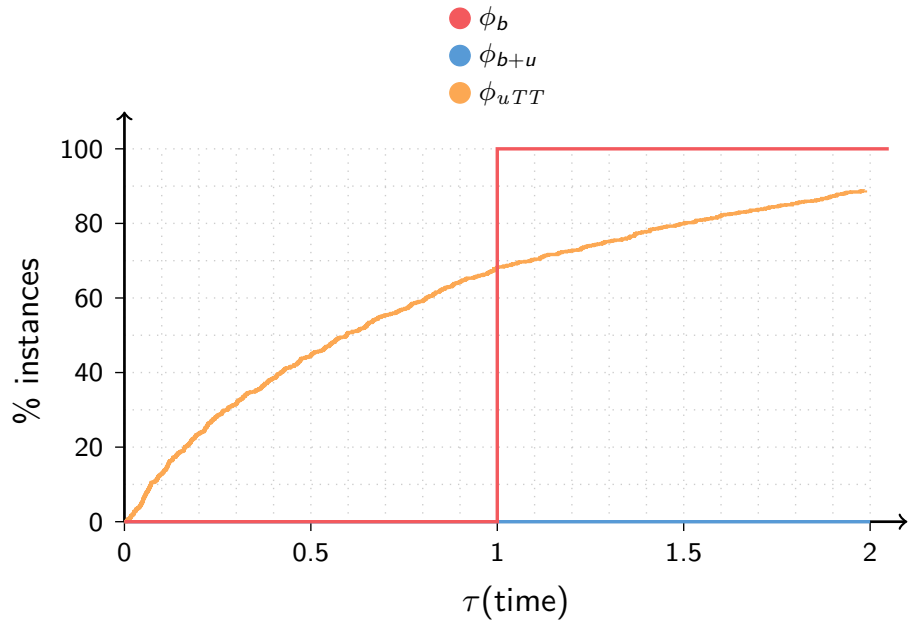



Figure 5.24: Performance profiles of the three models for the replay time on a sequence of nodes generated with the COS heuristics and $\phi_b$ as explained in Chapter 3.

nodes. This is especially true with the COS strategy that will tend to move up in the tree the variables that trigger failure. However, with the same search strategy but with different propagators – e.g., the unary resource with transition times introduced here – the sequence of nodes might have been completely different. For this reason, we present here results in a more "traditional" fashion.

We compute the best makespan that can be obtained with $\phi_b$ within 600 seconds, using the following binary static search strategy: fixed variable order, left branch assigns $s_i$ to $\text{est}_i$, right branch removes $\text{est}_i$ from the domain of $s_i$. Then, the time and number of failures required by each model to find this solution are computed. We filtered out instances for which the solution was found by $\phi_b$ in less than one second. From this perspective, the 10 best and worst results are reported in Tables 5.13 and 5.14, respectively. On the 10 best instances, the gains (the number of failures and time) are significant (sometimes two orders of magnitude). On the 10 worst instances, the times obtained with $\phi_{uTT}$ are similar to the results using the classical unary resource (i.e. $\phi_{b+u}$), while they are at worst around 6.4 times slower than the simple binary

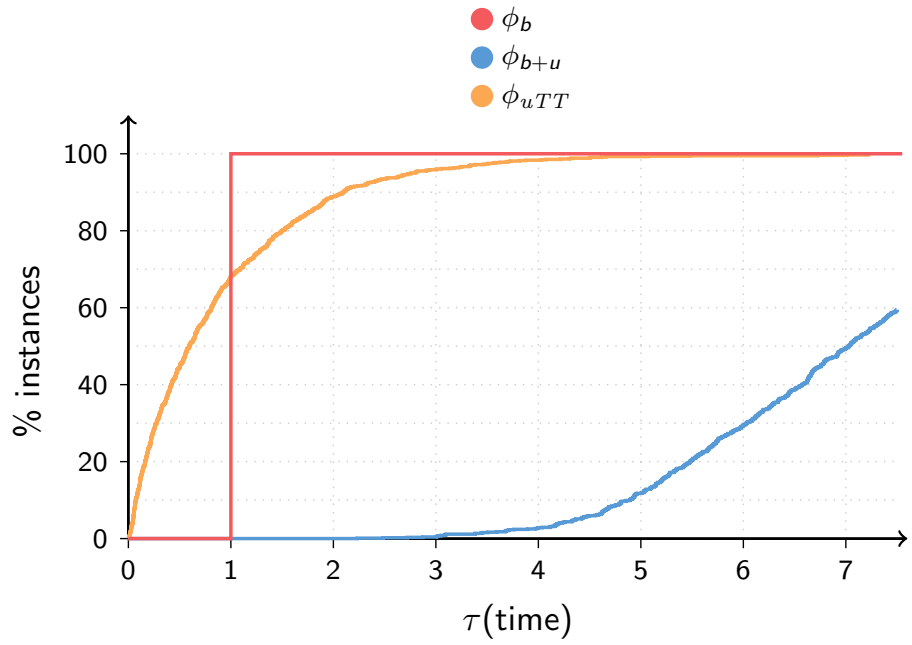



Figure 5.25: Long-term view of performance profiles from Figure 5.24.

decomposition (i.e. $\phi_b$). Note that on these instances, no additional pruning is achieved, or only a negligible additional pruning.

### 5.6.3 *Overview of Results*

This whole result and performance analysis allows us to draw the following conclusions:

IMPORTANT ADDITIONAL PRUNING
The propagator $\phi_{uTT}$ allows to achieve substantial additional pruning with respect to the binary decomposition $\phi_b$ on all instances.

CLASSIC UNARY RESOURCE PERFORMS BADLY
The propagator $\phi_{b+u}$ fails to achieve additional pruning in comparison with $\phi_b$ on most instances (rather small additional pruning on only 65% of the instances).

IMPORTANT TIME GAIN
The gain of time achieved by $\phi_{uTT}$ in comparison with $\phi_b$ can be important (at least twice as fast for 45% of the instances).

| Instance | $\phi_{uTT}$ | | $\phi_b$ | | $\phi_{b+u}$ | |
|---|---|---|---|---|---|---|
| | Time | #Fails | Time | #Fails | Time | #Fails |
| Best-1 | **1.12** | **2,442** | 117.92 | 980,330 | 432.07 | 911,894 |
| Best-2 | **2.11** | **744** | 182.27 | 1,127,272 | 999.79 | 1,127,272 |
| Best-3 | **0.24** | **449** | 17.63 | 168,466 | 62.27 | 168,466 |
| Best-4 | **3.90** | **5,593** | 187.93 | 889,079 | 534.20 | 602,591 |
| Best-5 | **2.96** | **1,709** | 126.61 | 584,407 | 829.25 | 584,407 |
| Best-6 | **11.59** | **885** | 340.32 | 332,412 | 1225.44 | 206,470 |
| Best-7 | **1.97** | **1,178** | 39.23 | 226,700 | 314.34 | 226,700 |
| Best-8 | **0.91** | **2,048** | 16.40 | 119,657 | 63.38 | 119,657 |
| Best-9 | **3.79** | **1,680** | 63.16 | 878,162 | 4.63 | 1,695 |
| Best-10 | **0.74** | **687** | 9.24 | 106,683 | 41.25 | 106,683 |

Table 5.13: Best time results for $\phi_{uTT}$ compared to $\phi_b$. The problem is to find a given makespan: the smallest makespan found in 600 seconds by $\phi_b$ using a binary static search strategy. Time is reported in seconds.

*Future Work*

The propagation procedure involved in this chapter has some limitations. The two most important ones are as follows:

- It relies on loose bounds of the ect of a set of activities.
- It is inefficient when considering *sparse* transition times between activities i.e., when the number of null transition times is large.

Several improvements could be considered to overcome the two limitations of the work presented in this chapter. Let us address some ideas that could potentially increase the impact of this work. We will first address ideas to overcome the loose bound on the ect of a set of activities, then we will discuss current research performed to deal with sparse transition times.

| Instance | $\phi_{uTT}$ | | $\phi_b$ | | $\phi_{b+u}$ | |
|---|---|---|---|---|---|---|
| | Time | #Fails | Time | #Fails | Time | #Fails |
| Worst-1 | 645.26 | **546,803** | **127.38** | **546,803** | 572.81 | **546,803** |
| Worst-2 | 954.77 | **164,404** | **174.63** | 164,437 | 1,108.43 | 164,437 |
| Worst-3 | 213.54 | **78,782** | **38.26** | 78,968 | 180.20 | 78,968 |
| Worst-4 | 147.55 | **164,546** | **26.42** | 164,576 | 175.69 | 164,576 |
| Worst-5 | 178.37 | **96,821** | **31.23** | **96,821** | 139.84 | **96,821** |
| Worst-6 | 11.15 | **8,708** | **1.94** | 8,745 | 11.87 | 8,745 |
| Worst-7 | 18.63 | **6,665** | **3.15** | 6,687 | 19.93 | 6,687 |
| Worst-8 | 85.84 | **61,185** | **14.24** | **61,185** | 65.12 | **61,185** |
| Worst-9 | 286.61 | **88,340** | **46.17** | **88,340** | 180.23 | **88,340** |
| Worst-10 | 189.37 | **208,003** | **29.55** | 209,885 | 157.33 | 209,885 |

Table 5.14: Worst time results for $\phi_{uTT}$ compared to $\phi_b$. The problem is to find a given makespan: the smallest makespan found in 600 seconds by $\phi_b$ using a binary static search strategy. Time is reported in seconds.

*Tightening the ect lower bound*

The Θ-tree and Θ-Λ-tree data structures allow to compute tighter bounds of the ect than the original versions from Vilím when transition times are considered. However, they have weaknesses that would benefit being investigated.

First of all, the update rules take transitions into account only based on the cardinality of the set of activities considered: $\underline{\mathrm{tt}}(k)$. Other works have considered a different approach such that the lower bounds of the ect of a set of activities is tighter. For example, Artigues et al. [AF08] propose to pre-compute all the exact TSPs for all the possible sets of activities (the exact TSP procedure defined in this chapter). However, the number of all the possible sets of activities is exponentially large; for $n$ activities, there are $2^n$ different possible sets. Nevertheless, Artigues proposes a dynamic program that computes all the TSPs (hence all the minimal times spent by transitions) for $n$ activities with a time complexity in $\mathcal{O}(n^2 \cdot 2^n)$. Furthermore, we can consider that the value

for the TSP of a given set of activities is stored in memory on an integer, nowadays stored on 4 bytes. This means that for $n$ activities, one would need $4n \cdot 2^n$ bytes to store these values (for each possible set, we have to store the smallest TSP considering every time that one activity from the set is the first one). To give an overview of how badly such decision scales, we have reported both the number of operations and the memory required to store such data in Table 5.15. Even if some

| k | Number of Operations | Scaled Memory Size |
|---|---|---|
| 5 | $400 \cdot 10^0$ | 40 b |
| 10 | $51.2 \cdot 10^3$ | 2.56 kb |
| 15 | $3.68 \cdot 10^6$ | 122.88 kb |
| 20 | $209.71 \cdot 10^6$ | 5.24 Mb |
| 25 | $10.48 \cdot 10^9$ | 209.71 Mb |
| 30 | $483.18 \cdot 10^9$ | 8.05 Gb |
| 35 | $21.05 \cdot 10^{12}$ | 300.64 Gb |
| 40 | $879.60 \cdot 10^{12}$ | 11 Tb |
| 45 | $35.62 \cdot 10^{15}$ | 395.82 Tb |
| 50 | $1.41 \cdot 10^{18}$ | 14.07 Pb |

Table 5.15: Number of operations and amount of memory required to compute and store the TSPs representing the transitions inside all the sets of $k$ activities, as proposed in [VB02, AF08].

supercomputer were able to perform the tremendously huge amount of operations in less than billions of times the age of the universe, one would still need to buy a lot of RAM memory to be able to store all the computed data. As one can see, this approach is not meant to scale on large instances and therefore does not fit our needs. We have considered instances with up to 2000 activities (i.e., $k = 2000$) in our benchmarks.

Unfortunately, even if we had access to the exact value of the smallest transition TSP of a set of activities, we could not use it in the $\Theta$-tree and $\Theta$-$\Lambda$-tree. Indeed, those structures rely on the separation of the computation of $\text{ect}_{\text{Left}(v)}$ and $\text{ect}_{\text{Right}(v)}$. We could not combine those together if we considered exact TSP values. Indeed, if we consider two

disjoint sets of activities $\Omega_1$ and $\Omega_2$ ($\Omega_1 \cap \Omega_2 = \varnothing$), we have the following inequality:

$$\text{TSP}(\Omega_1 \cup \Omega_2) \leq \text{TSP}(\Omega_1) + \text{TSP}(\Omega_2) + \min_{A_i \in \Omega_1, A_j \in \Omega_2} \text{tt}_{ij}$$

This equation forbids to *merge* the TSPs of two subsets of activities to compute the ect of the union of those latter. Other options should be considered to replace the lower bound obtained only on the cardinality of a set of activities.

Finally, the total sum of transition times by cardinality taken into account in the root is a sum of bounds of sets of small cardinality. Indeed, our update rules include only the lower bound based on the cardinality of the *right* subset of activities. This means that the largest cardinality considered will be $n/2$ for a tree containing $n$ activities. For example, if we consider a complete $\Theta$-tree of 8 activities, the maximal sum of the terms taking transition times into account is as follows:

$$\underline{\text{tt}}(4) + \underline{\text{tt}}(2) + \underline{\text{tt}}(1) \leq \underline{\text{tt}}(7)$$

However, this decomposition into subsets of smaller cardinalities is mandatory to maintain the low time complexity obtained by incremental computation. Some investigations should evaluate the possibility of obtaining less divisions into subsets, hopefully obtaining a tighter bound for the transitions included in a set of activities.

*Robustness Over Sparse Transition Times*

Our lower bounds for the transition times occurring in a set of activities relies only on the cardinality of the set. In the case of a sparse transition time matrix, such lower bounds can be pretty loose for small cardinalities. Let us consider a small example of six activities with the sparse transition matrix from Figure 5.26. The associated computed lower bounds are also displayed in Figure 5.26. First of all, we can see that the relaxations used to compute the lower bounds are not close to the real TSP values when a lot of transitions are null. Then, even when considering the real TSP values obtained, they are very loose. As the $\Theta$-tree and $\Theta$-$\Lambda$-tree structures both rely on these lower bounds by cardinality, the computed ect$^*$ of a set of activities will be loose.

However, if we carefully look at the transition matrix in Figure 5.26, we can see that we could group activities into "families". One could group activities into families as follows: $A_1, A_2 \in f_1$, $A_3, A_4 \in f_2$ and

$$\mathcal{M} = \begin{pmatrix} 0 & 0 & 4 & 4 & 6 & 6 \\ 0 & 0 & 4 & 4 & 6 & 6 \\ 5 & 5 & 0 & 0 & 8 & 8 \\ 5 & 5 & 0 & 0 & 8 & 8 \\ 7 & 7 & 9 & 9 & 0 & 0 \\ 7 & 7 & 9 & 9 & 0 & 0 \end{pmatrix}$$

| LB | $k$ | | | | | |
|---|---|---|---|---|---|---|
| | 0 | 1 | 2 | 3 | 4 | 5 |
| Sum of Min. Trans. | **0** | **0** | **0** | **0** | 0 | 0 |
| Min. Weight Forest | **0** | **0** | **0** | **0** | **4** | **10** |
| Dyn. Prog. | **0** | **0** | **0** | **0** | 0 | 0 |
| Min. Ass. | **0** | **0** | **0** | **0** | 0 | 0 |
| Lag. Relax. | **0** | **0** | **0** | **0** | 0 | 0 |
| $\underline{\text{tt}}(k) = \max$ | 0 | 0 | 0 | 0 | 4 | 10 |
| Exact TSP | 0 | 0 | 4 | 4 | 11 | 11 |

Figure 5.26: Example of transition times and lower bound of transitions within a set of activities by cardinality.

$A_5, A_6 \in f_3$. As transitions between activities inside each family are null, it could be useful to count the cardinality of *different families* inside a set of activities instead of counting the number of activities. Indeed, the transition matrix from Figure 5.26 can be transformed into a transition matrix between families as shown in Figure 5.27. We can see that the lower bounds obtained on the transitions between families are much tighter than those obtained with the original transition matrix. Let us consider a small set of three activities $\Omega = \{A_1, A_3, A_5\}$. The exact transition TSP lower bound considering the cardinality over activities would give $\underline{\text{tt}}(2) = 4$. A stronger lower bound can be obtained if we count the number of different families (here three different families): $\underline{\text{tt}}(2)^{\text{families}} = 11$. Counting the number of different families instead of the number of activities would require adaptations of the extended $\Theta$-tree and $\Theta$-$\Lambda$-tree structures introduced in this chapter. However, these modifications could bring a huge additional pruning. A work on such improvement has been submitted to the CP2016 conference and the results so far show that the gain in terms of pruning can be huge when families can be considered.

The families from the example in Figures 5.26 and 5.27 can be deduced easily as they have the two following properties:

- There are only null transitions between activities from a same family.
- Activities from a given family all have the same transition to and from any activity from another family.

$$\mathcal{M}^{\text{families}} = \begin{pmatrix} 0 & 4 & 6 \\ 5 & 0 & 8 \\ 7 & 9 & 0 \end{pmatrix}$$

| LB | $k$ | | |
|---|---|---|---|
| | 0 | 1 | 2 |
| Sum of Min. Trans. | **0** | **4** | 9 |
| Min. Weight Forest | **0** | **4** | **10** |
| Dyn. Prog. | **0** | **4** | 9 |
| Min. Ass. | **0** | **4** | 9 |
| Lag. Relax. | **0** | **4** | 9 |
| $\underline{\text{tt}}(k)^{\text{families}} = \max$ | 0 | 4 | 10 |
| Exact TSP | 0 | 4 | 11 |

Figure 5.27: Example of transition times and lower bound of transitions between families by cardinality.

However, there could be cases where one or both of these properties are not respected even though the problem contains null transitions or transitions of widely different importances. From there came the idea that a clustering technique could be applied on the initial matrix. The obtained clusters could then be considered as families and the family count based reasoning could be applied. If a cluster contains non-null transitions between its internal activities, one could also count the number of activities from that cluster and use a cardinality reasoning on transitions internally to the cluster. Note that there could be several levels of clusters for large transition matrices. The larger clusters could be made to take large transitions into account. Inside a cluster, there could be other sub-clusters that would include smaller – but still larger than the average – transitions. This recursive clustering would not be limited to two levels, but could include more.

### CONCLUSION

In this chapter, we have proposed an extension of the classic unary resource propagation algorithms in order to take transition times into account. We have proposed to compute a lower bound of the time taken by transitions occurring between activities from a set $\Omega$. This allows to compute a tighter lower bound for $\text{ect}_\Omega$. We have proposed several methods to compute these lower bounds and we have shown that the

obtained lower bounds were close to the smallest TSPs with $k$ activities. To achieve an efficient computation of $\text{ect}_\Omega$, we have proposed to integrate these lower bounds in extensions of the $\Theta$-tree and $\Theta$-$\Lambda$-tree structures. These extended structures can then be used in slightly modified versions of classic unary propagation algorithms; namely Overload Checking, Detectable Precedences, Not-First/Not-Last and Edge Finding. From these algorithms, we have proposed an internal propagation loop that also integrates the binary decomposition of the constraint. This new obtained propagation procedure has the advantage that it can be used conjointly with any other constraint and that it is completely independent from the objective function to optimize.

The results of our approach have highlighted that the additional pruning achieved by this propagation can dramatically reduce the number of nodes. On the other hand, we have also shown that the classic unary resource propagation procedure (that does not include transition times) rarely proposes a substantial gain in terms of pruning. The additional gain from our propagation procedure allows to solve problems faster, sometimes dramatically faster. However, the overhead of this propagation procedure is non-negligible when compared to the one of the binary decomposition propagation. Hence, the propagation procedure we have introduced is slower when it fails to achieve substantial pruning (maximum 7.5 times slower when a small additional pruning is achieved).

The potential of the propagation procedure introduced in this chapter is quite large. We have proven that it could achieve a large pruning with a low time complexity. However, this research is just the beginning; several promising ideas could potentially increase the achieved pruning obtained. There could be more than one single PhD thesis centered on this constraint. Several other ideas have not been discussed but are worth being mentioned: using $\Theta$-tree to compute TSPs, use $\Theta$-tree to compute a lower bound on the makespan, ...

It is our strong belief that this constraint can be improved in many ways and that its true potential has yet to be revealed.

# Part III

# PATIENT SCHEDULING IN RADIOTHERAPY

# 6

# PROTON THERAPY PATIENT SCHEDULING

*To keep the body in good health is a duty... otherwise we shall not be able to keep our mind strong and clear.*

—Buddha

*Occam's Razor. The simplest explanation is almost always somebody screwed up.*

—Gregory House, *House, M.D.*

*Healing is a matter of time, but it is sometimes also a matter of opportunity.*

—Hippocrates

*Music is therapy. Music moves people. It connects people in ways that no other medium can. It pulls heart strings. It acts as medicine.*

—Macklemore

*A healthy outside starts from the inside.*

—Robert Urich

Proton Therapy (PT) is a recent technique in which a beam of protons is used to treat cancer. It offers many advantages over classic radiotherapy techniques. Even if this technique is recent and still under study, treatments performed so far have shown promising results. The management of the schedule of patient treatment is complex and subject to many constraints. To date, no computer procedure exists to optimize the schedule of patient treatments within PT centers. IBA (Ion Beam Applications)[1] has proposed to study this particular problem. The work presented in this chapter has thus been realized in collaboration with IBA.

The Proton Therapy Problem (PTP) consists in optimizing the schedule of patient workflows within a PT treatment center. This problem can be challenging since it considers the optimization of several non-trivial objectives. These objectives can depend on the specific configuration of the PT center considered. We propose an approach using Scheduling in CP allowing to easily adapt the model to the changing requirements of the physicians regarding the schedule.

A second problem that was studied during this thesis was the Ten Weeks Ahead Appointment Schedule Problem (TWAASP). This problem aims at finding an optimal schedule for appointments corresponding to the treatment sessions a patient has to go through. When a patient is diagnosed with cancer, he has to follow a specified amount of treatment sessions. Each treatment session has its own parameters and corresponds to a specific patient workflow, with its various steps, durations and resource demands. The problem aims at determining when these sessions will be placed in the PT center global schedule such that an objective function is optimized.

We first give an overview of the first considered problem in Section 6.1. Section 6.2 describes the Scheduling in CP resolution of the problem. Finally, Section 6.3 presents the Ten Weeks Ahead Appointment Schedule Problem (TWAASP) and Section 6.4 introduces the CP approach used to solve it.

### RELATED PUBLICATIONS

[Dej13] Cyrille Dejemeppe. "Alternative Job-Shop Scheduling For Proton Therapy." In: *CP Doctoral Program 2013* (2013), p. 67.

1 http://www.iba-worldwide.com/

## 6.1 THE PROTON THERAPY PROBLEM

Proton Therapy (PT) [Shi+79] is a technique used in the treatment of cancer. It uses a beam of protons to irradiate diseased cells. As other forms of radiotherapy, it sends ionizing energy particles (here protons) to the diseased tissues. A particle accelerator is used to generate the proton beam. The protons damage the DNA of the cells, eventually causing their death or limiting their ability to procreate. Due to their high rate of cell division and their reduced ability to repair DNA, the cancerous cells are more vulnerable than healthy cells to the attack of ionizing particles on DNA.

As protons are heavier than photons, they spread less into surrounding tissues, reducing damages on healthy cells surrounding targeted tumors. Protons, as all charged particles, have a rapid energy loss in the last millimeters of penetration into human tissues. This allows to sharply define a maximal distance of penetration since this distance is directly related to the initial energy of charged particles. Furthermore, only the last millimeters of this penetration depth receive the maximal dose. As stated in [Lev+05], this phenomenon of a localized sharp peak of dose is known as *Bragg peak*. To irradiate a tumor, one has to vary beam energy and intensity to obtain the desired dose over the tumor volume. Tissues located along the ray but at lower depth than Bragg peak receive a reduced dose. Tissues located at higher depth than Bragg peak receive no dose at all. The main benefits of PT over classic radiotherapy techniques are that tissues behind the tumor are not irradiated and that reduced damages are caused to surrounding tissues.

PT requires a particle accelerator that can be expensive. It also demands several additional pieces of equipment with large space requirements. This is one of the reasons for which nowadays there are only few PT centers worldwide. According to [Gro12], in 2012 there existed 40 PT centers in operation worldwide. They are located in North America, Europe, Asia and South Africa. The types of cancer treated by PT include ocular tumors, skull base and para-spinal tumors, unresectable sarcomas, pediatric neoplasms and prostate cancer. This technique is still under study but according to [Lev+05], treatments performed so far have shown promising results.

### 6.1.1 *PT Treatment Center*

To date there are 40 PT centers in operation around the world. While these centers differ on many points, most of them share a common structure. The structure that is described here is inspired by the PT center in Essen, Germany. There can be many variants of this structure and most of the centers are variants of a base model (as those designed by IBA[2]).

PT centers contain a particle accelerator (cyclotron or synchrotron) which produces the proton beam. This particle accelerator can be set up to a given intensity and energy level to deliver the dose desired for a given patient. The other main rooms of PT centers are the following ones:

PREPARATION ROOM
: In order to make sure the tumor is hit by the proton beam and as few healthy tissues as possible are damaged, patients are immobilized. To do so, they are bound to a treatment board, most of the time aided with specific immobilization equipment. It is in the preparation room that patients are immobilized before treatment. It is also in this room that they are released from the immobilization equipment after treatment. For matters of precision and ease, every patient is assigned to his own immobilization equipment. Dressing rooms are attached to these preparation rooms to allow patients to undress and dress respectively before and after treatment. There are often several preparation rooms in a PT center in order to allow several patients to prepare simultaneously.

CORRIDOR
: Corridors link rooms of the PT center. Congestion has to be taken into account because of the limited space a corridor can provide. Indeed, trolleys are used to move immobilized patients. The width of corridors allow at most two trolleys to cross each other. Having trolleys with patients waiting in a corridor for a treatment room to be available is a common situation.

SCAN ROOM
: In order to determine the location and size of the tumor, patients have to be scanned. These informations obtained by the scanner allow to calibrate the beam and the position in which the patient will be

2 For more details about the services IBA provides for PT, refer to [IBA12].

irradiated. The scanner is also used to track the evolution of the tumor through treatment.

ANESTHESIA ROOM
Some patients are treated under anesthesia. The anesthesia is performed in a dedicated anesthesia room. An awakening room is attached to the anesthesia room in order to monitor the awakening of patients and make sure anesthesia took place as planned.

TREATMENT ROOM
According to the type of tumor, a patient will be treated in a given treatment room. There are different types of treatment rooms (e.g., fixed beam rooms, single and double scattering rooms, gantry rooms, etc.). In these treatment rooms, a huge device called *snout* is used to shape the proton beam. The same snout can be used for patients with the same type of tumor. This big device has to be changed between two treatments of different type of tumors. In most centers, there is one beam line and only a single treatment room can deliver the proton beam to a patient at a given time. It takes some time to install a patient in a treatment room. To save time, while a patient receives the proton beam, other patients are installed in other treatment rooms. When a patient is installed and ready to receive his dose, a demand to get the proton beam is sent to the control room and the patient waits until he has received the required dose. Switching the proton beam from one treatment room to another takes some time (from 1 to 2 minutes).

CONTROL ROOM
The control room is used to assign the proton beam to a treatment room and to adjust its energy level and intensity. When a treatment room queries the proton beam, its request is inserted in a FIFO[3] queue. When a beam request reaches the front of the queue, the treatment room who queried the beam receives it (as soon as the current dose delivery is over) and the query is removed from the queue.

Figure 6.1 is a schema of the structure of a PT center. There are several treatment rooms and their types may differ (gantry, fixed beam, etc.) or may be redundant (some PT centers only cure a single type of cancer). As explained before, there are several preparation rooms and scanning rooms to allow respectively simultaneous patient preparations and scans.

3 First In First Out

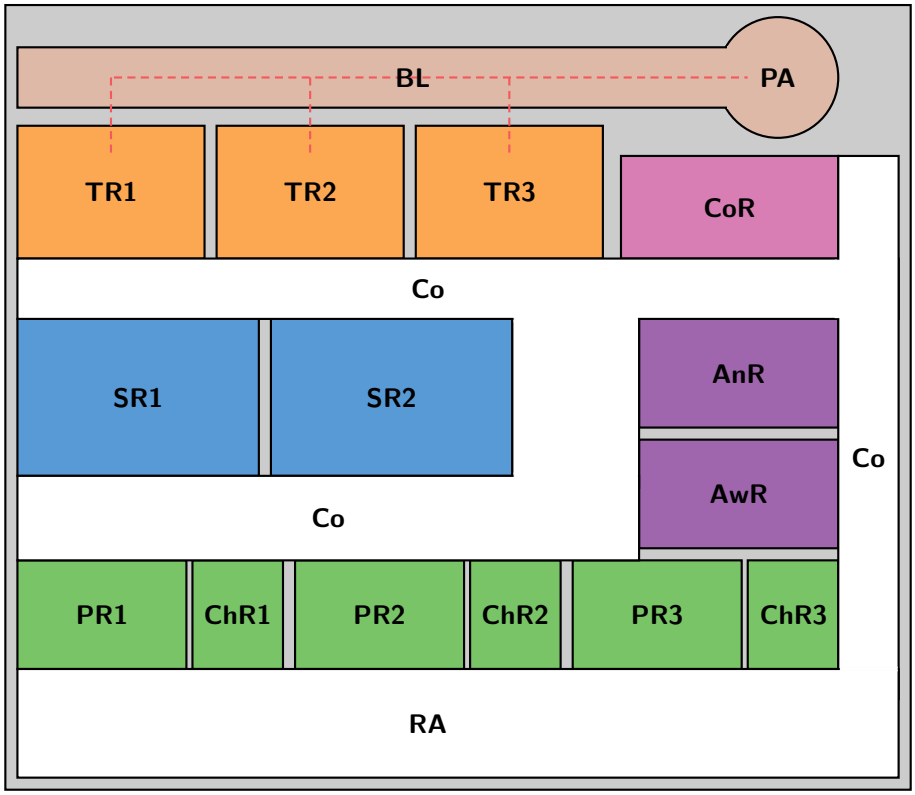



Figure 6.1: Schema of a PT center and its main rooms. The dotted red line represents the proton beam. **PA** is the Particle Accelerator (cyclotron or synchrotron). **BL** is the Beam Line. **TR** holds for Treatment Room. **CoR** is the control room. **SR** holds for Scanning Room. **AnR** and **AwR** are respectively the Anesthesia Room and the Awakening Room. **PR** and **ChR** holds respectively for Preparation Room and Changing Room. **Co** holds for Corridor. **RA** is the Reception Area.

### 6.1.2 *Patient Workflow*

When a patient is diagnosed with cancer, a given number of sessions is determined for his treatment. His workflow (sequence of steps he has to follow during one treatment session) is also determined during diagnosis. The workflow assigned to a patient is defined according to the type of cancer, the location and size of the tumor, along with other factors. The workflow of patients may differ even if they have the same type of cancer. For example, a child workflow systematically begins with an anesthesia while this is rarely the case for an adult

workflow. The time spent at every step of the workflow of a patient may be approximated but it varies due to human factors. Even if two patients have the same workflow, the time they spend at every step of this workflow may differ. In our work, we have considered that the workflow assigned to each patient is known and determined (including the time spent in every step of his workflow).

### 6.1.3 *Staff Workflow*

The staff working in a PT center has also to be taken into account in our model. The staff members we are interested in are those whose job imply they will intervene in patient workflows. Some staff members are *static* in the sense they are assigned to given rooms and don't accompany patients through several steps of their workflow (e.g., staff in scan room). Other staff members are needed to guide patients through corridors to the different rooms in their workflow (scan, treatment, anesthesia, etc.). Finally some staff members are needed to perform technical tasks such as changing a snout or managing the repartition of the beam from the control room.

### 6.1.4 *PTP Scheduling Objectives*

The optimization of the daily schedule of patients can be performed on several criteria. Here is a list the most important of those:

PATIENT THROUGHPUT
: The patient throughput represents the number of patients that can be treated on a single day. If more patients can have their treatment session in a day, then more patients could have their whole treatment in a PT center. One objective is thus to maximize the patient throughput.

PATIENT COMFORT
: During a treatment session, patients have to be immobilized sometimes in very uncomfortable positions. The amount of time patients spent immobilized (i.e., the amount of time needed by the overall workflow of a patient) must then be minimized in order to improve patient comfort or, equivalently, reduce patient discomfort.

STAFF WELFARE
: Staff welfare can be expressed in a scheduling problem as a smooth-

ing of the schedule. One of the optimization criteria becomes the minimization of downtimes for staff members.

One of the goals of PTP is to combine these objectives. We would then provide a set of solutions whose quality can be quantified according to each criterion described above. This set of solutions would be an estimation of the Pareto set of our problem. A human user would then select one of the schedules proposed in the Pareto set.

## 6.2 A SCHEDULING MODEL FOR PTP

In this section, we present a scheduling model for PTP. This simple model has been solved with Constraint Programming (CP). First, we explain the model, then we describe the resolution strategy applied to solve it.

### 6.2.1 *Model*

The main components of our scheduling model will be described here: activities, resources, constraints, and objectives.

ACTIVITIES

Every workflow of a patient is represented as a job and every step of the workflow is represented as an activity in that job. As the treatment steps from a patient's workflow cannot be interrupted, we consider the activities as non-preemptive. We also consider that the durations of the activities are known for every patient whose treatment session has to be scheduled.

RESOURCES

The two main resources considered in this work are the PT center rooms and treatment tables. PT centers generally contain several instances of some treatment rooms (allowing parallel flow of patients). These rooms can be modeled with cumulative resources whose capacity corresponds to the number of instances of the treatment room considered. Other rooms however, are present in only a single instance. These latter are modeled with unary resources. The treatment tables are available in several instances and are also represented with cumulative resources. Finally, the proton beam is modeled with a unary resource as it cannot be split and shared between several treatment rooms simultaneously. The simple decomposition of activities into

the different steps of a treatment session is not enough here. Indeed, some resources, such as treatment tables, have to be held for a succession of activities and cannot be released during idle times between these activities. To model this, we use *super* activities. A super activity is an activity containing a succession of activities such that the starting time of the super activity corresponds to the starting time of the first activity it contains and its ending time corresponds to the ending time of the last activity it contains. With such formulation and the usual resource constraints, it is not possible for a resource used by a super activity to be released between the activities the super activity contains. The members of the staff are also taken into account in this model. They are modeled as cumulative resources, one cumulative resource for each type of staff member e.g., physicians, nurses, technical supervisors, etc.

CONSTRAINTS

The different steps in the treatment session of a patient must take place in a fixed order. There are thus precedences ordering the activities inside each job as they represent treatment sessions for specific patients. Then, for each of the resources we have described earlier, there is a cumulative or unary resource constraint corresponding to the kind of resource considered. The beam is modeled with a unary resource and transition times are also taken into account. Indeed, some delay – i.e., transition time – is required to switch the beam between treatment rooms. Note that this delay depends on the two rooms considered and the transition matrix will thus contain various values. A similar constraint models the amount of time required to change the snout between two successive patients with different tumor types. Another constraint imposes that two activities cannot be separated by more than a maximal amount of time. This is done to ensure that patients under anesthesia will not be put to sleep for a too long period.

OBJECTIVES

We consider a single objective in this simple model: the maximization of the patient throughput. We model this with a minimization of the makespan over the treatment sessions that have to be scheduled. Indeed, if a given number of treatment sessions can be performed in a shorter amount of time, then there could be more additional treatment sessions added on the same day. The second objective considered is to maximize the comfort of patients. Patients whose treatment

takes a lot of time, because of idle times between treatment steps, will experiment discomfort. Similarly, if patients are rushed through their treatment steps without some small idle times, they will also experiment discomfort. Hence we consider that the whole treatment of a patient should contain some idle times neither too long nor too short. This is modeled by minimizing the difference between the total time taken by a job (i.e., a patient treatment session) and a predefined value that is supposed to represent the optimal time for a treatment session. The third objective is the maximization of staff welfare. To reduce stress and allow normal shifts for staff members, huge peaks of activity should be avoided. The definition of peak of activity being confusing, we try to smooth as much as possible the demand of staff members. This is modeled with the usage of the cumulative resources representing staff members. We sum the difference of the usage of these resource between successive steps of time.

Formally, the model described above is as follows:

$$
\begin{aligned}
\text{minimize} \quad & w_1 \cdot \texttt{makespan} && (6.1)\\
& + \; w_2 \cdot \sum_j (e_{j,n} - s_{j,1} - \text{prefDur}_j)^2 && (6.2)\\
& + \; w_3 \cdot \sum_{sr} \sum_t (\text{usage}(sr, t) - \text{usage}(sr, t-1))^2 && (6.3)\\
\text{such that} \quad & \forall j, \forall i : A_{j,i} \ll A_{j,i+1} && (6.4)\\
& \forall ur : \texttt{UnaryResource}\,(\mathcal{A}, U_{ur}) && (6.5)\\
& \texttt{UnaryResourceWithTT}\,(\mathcal{A}, U_{\text{beam}}, \mathcal{M}_{\text{beam}}) && (6.6)\\
& \texttt{UnaryResourceWithTT}\,(\mathcal{A}, U_{\text{snout}}, \mathcal{M}_{\text{snout}}) && (6.7)\\
& \forall cr : \texttt{CumulativeResource}\,(\mathcal{A}, U_{cr}, C_{cr}) && (6.8)\\
& \forall ja : (e_{ja,n} - s_{ja,1}) < \text{durAnesthesia}_{ja} && (6.9)
\end{aligned}
$$

where $\mathcal{A}$ is the set of all the activities, $A_{j,i}$ is the activity at position $i$ of job $j$, $ur$ denotes a unary resource and $U_{ur}$ is a vector defining for each activity whether or not it uses $ur$, $cr$ is a cumulative resource, $U_{cr}$ is a vector defining for each activity the usage of $cr$ and $C_{cr}$ is the capacity of $cr$, $w_1$, $w_2$ and $w_3$ are positive weights determining the importance of each objective, $\text{prefDur}_j$ is the desired duration of the treatment session of patient $j$ (i.e., job $j$), $sr$ denotes staff resource (a cumulative resource representing a staff crew), $\text{usage}(r, t)$ is the usage of the resource $r$ at time step $t$, `unaryResourceWithTT` is a unary resource with transition times as described in Chapter 5, $\mathcal{M}_{\text{beam}}$ and $\mathcal{M}_{\text{snout}}$

represents the transition times between activities using respectively the beam and snouts, *ja* represents a patient with anesthesia (a job containing activities in relation with anesthesia) and durAnesthesia$_{ja}$ is the maximal duration of anesthesia for patient *ja*.

This model could be extended to add more constraints should the configuration of a PT center require it. All the constraints and parameters of our problem allow us to define our model as a cumulative job-shop problem.

### 6.2.2 *Simplifications and Potential Improvements of the Model*

Several simplifications of the model have been performed. First of all, several quantities have been modeled with cumulative resources. This is the case for the treatment rooms, the treatment tables and the staff members. Indeed, the cumulative resource does not forbid rectangle, i.e., activities, overlapping, it only enforces that the sum of the usage of activities at any point of time does not exceed its capacity. Hence a situation might occur where the cumulative resource constraint is respected but the actual schedule enforces an activity to start on a given resource and end on another one (e.g., a patient begins his treatment activity in a given room and has to end it in another treatment room – which is not allowed). Hence, a finer model would use alternative resources, where a patient is *assigned* to a single treatment room and a single treatment table.

This simplification has been made because the quantities modeled with cumulative resources are most of the time over-capacitated. In the instances that we have run, it only has happened a few time that the usage of quantities modeled by cumulative resources had reached its capacity. Furthermore, when resource usage met the capacity, it was always possible to find an arrangement of the rectangles representing the activities on the resource such that they did not overlap. Therefore, to keep a concise model, we have decided to consider that a post-processing step was able to arrange the activities at peak usage such that they did not overlap. Of course, a more refined model should consider alternative resources because in some instances, this post-processing step could not be feasible.

In this model, the second and third objectives from Equations (6.2) and (6.3) are sums of squared differences. This could have been modeled with a sum of absolute values of the differences but we have

chosen the squared version to favor multiple small differences instead of a few large ones.

The third objective from Equation (6.3) has been modeled with a sum of squared differences of resource usage between consecutive time steps. It could also have been modeled with the smooth cumulative constraint from Beldiceanu et al. [Bel+15]. We have however not integrated this objective that maximizes the comfort of staff members. Indeed, to our knowledge it would be hard to implement an *efficient* propagator such as the smooth cumulative constraint [Bel+15] to bound the objective value. Due to a lack of time we unfortunately have not been able to design and implement such a propagator.

### 6.2.3 *Resolution*

We have used a CP approach to solve this advanced model. The model from Section 6.2.1 was translated into a classic CP scheduling model $\text{CSP}(X, D, C, O)$.

We have used a classic *setTimes* heuristics as search strategy to solve our problem. As the search space is quite large, we coupled our resolution with a LNS strategy. We have considered several relaxations for this strategy:

IMPOSE START VARIABLES
: Maintain start variables values except for a random $x$% of them. Typically, we have set $x \in [50, 90]$ i.e., between 50% and 90% of the start variables were relaxed.

IMPOSE PRECEDENCES
: Maintain precedences between activities except for a random $x$% of them. To detect the precedences, we have considered *all* the possible pairs of activities $A_{j,i}$, $A_{k,l}$ such that $A_{j,i} \ll A_{k,l}$; if the detected precedence was not imposed by the original model, we have randomly relaxed it with a $x$% probability. Typically, we have set $x \in [10, 35]$ i.e., between 10% and 35% of the precedences were relaxed.

IMPOSE PRECEDENCES ON RESOURCES
: Maintain precedences between activities using the same resource except for a random $x$% of them. To detect the precedences, we have considered *all* the possible pairs of activities $A_{j,i}$, $A_{k,l}$ using the same resource – either cumulative or unary – such that $A_{j,i} \ll A_{k,l}$; if the detected precedence was not imposed by the original model, we have

randomly relaxed it with a $x$% probability. Typically, we have set $x \in [10, 35]$ i.e., between 10% and 35% of the precedences on each resource were relaxed.

COMBINATIONS OF RELAXATIONS

From the three relaxation strategies described earlier, we have decided to combine them into a single relaxation. Hence, at each relaxation, we have applied the three relaxation strategies described earlier together while modifying their respective relaxation probability.

We have also decided to adapt our LNS strategy according to the results obtained. As explained in Chapter 1, the search for each relaxation is limited by a maximal number of failures. When this maximal number of failures is reached, the search stops and a new LNS iteration begins. During the first relaxations of the LNS strategy, the objective value is *far* from the optimum. On the other hand, after a sufficient amount of relaxations, the objective value should be closer to the optimum. Hence, it is easier to find a feasible solution in early LNS relaxations while it is harder in later relaxations. For such reasons, we have decided to vary the maximal number of failures allowed for each relaxations. When an LNS relaxation is successful, we decrease this maximal number of failures; oppositely, when a relaxation fails, we increase the maximal number of failures allowed. Obviously we still have to bound this maximal number of failures to keep a good trade-off between intensification (trying to find a solution for *one* particular relaxation) and diversification (trying *several* relaxations). Formally, the max number of failures was updated as follows:

$$\texttt{nFails}_{i+1} \leftarrow \begin{cases} \max\left(\alpha \cdot \texttt{nFails}_i, \texttt{nFails}_{\min}\right) & \text{new solution found} \\ \min\left(\beta \cdot \texttt{nFails}_i, \texttt{nFails}_{\max}\right) & \text{no new solution} \end{cases}$$

where $\texttt{nFails}_i$ is the maximal number of failures allowed at LNS iteration $i$, $\alpha$ is a constant such that $0.5 \leq \alpha \leq 1$, $\beta$ is a constant such that $1.0 \leq \beta \leq 2.0$ and $\texttt{nFails}_{\min}$ and $\texttt{nFails}_{\max}$ are respectively the minimal and maximal number of failures allowed for a single LNS iteration.

The choice of these relaxations was motivated by experience and common knowledge about what relaxation works well in practice in the context of scheduling problem. As for the various parameters of the relaxations, they were fixed with values that have proven to work well in practice on the instances that we had generated. To do so, we

have solved several times our problem while varying the relaxation chosen, its parameters, the seed of the random generator and the instances. We have gathered this information but we were unable to find a given relaxation strategy that clearly dominated all the other ones. However, these experiments have allowed us to determine the various ranges of parameters described earlier that provided the best results for each relaxation. These relaxations could have been selected with several automated relaxation strategies as the ones described in Chapter 1. We have however not implemented such framework in the case of this work.

*Results*

We have developed a random instance generator for PTP. We were unfortunately not able to test our model on real historical data. Nevertheless this generator was developed to generate patient mixes and PT center configurations that are supposed to be very close to real situations. We have generated 300 instances of sizes going from 20 to 80 patients. The PT center that we have visited was a large center and was treating around 50 patients daily. We have performed some tests while varying the weights on the objectives to give more or less importance to minimization of patient throughput or patient comfort. On the generated instances, for all the objective weight configurations we have tested, the solver was able in less than 10 seconds to find an initial feasible schedule. With another period of 10 seconds, the solver increased the quality of our schedule by about 25%. Furthermore, when the solver has run for a longer amount of time, it has been able to find solutions of even higher quality even if the gains would be very small.

In Figure 6.2, we present the evolution of the makespan ratio – the makespan divided by the best makespan observed – through time when applying successive LNS iterations. On this figure, the relaxation applied imposes the assignation of values to the start variables while relaxing a proportion of them. The relaxation probability tested here are 50%, 75% and 90%. We see clearly on this picture that this parameter has a large influence on the resolution time needed to reach a given objective value. Furthermore, some relaxation probabilities are unable to discover the best solution found. Indeed, as too few start variables are relaxed, the CSP is too constrained and therefore cannot find new solutions. Several other experiments were performed while

varying the parameters of LNS relaxations, we however have decided not to report them here.

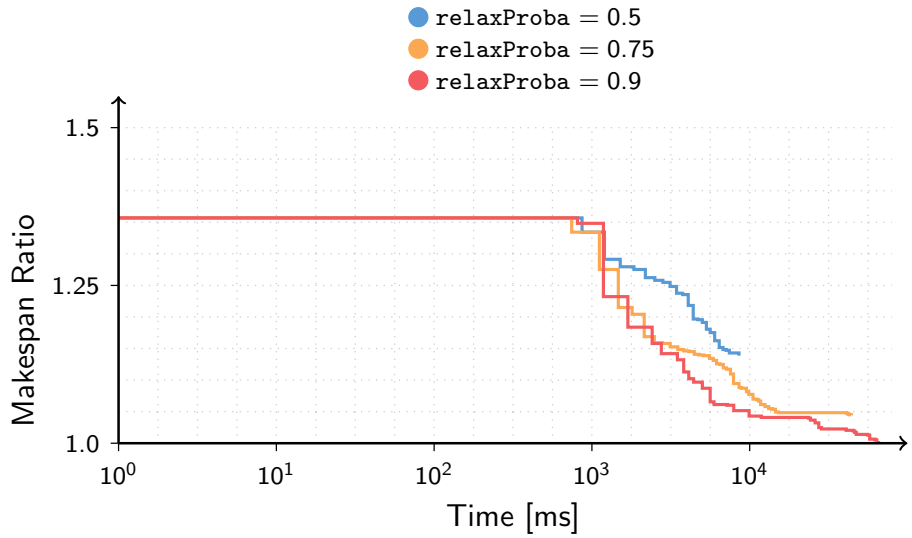



Figure 6.2: Evolution of the makespan ratio (makespan divided by best makespan) through time when applying LNS relaxations with various relaxation probability on an instance of 70 patients. The LNS relaxations impose start variables while relaxing a given percentage of them.

### 6.2.4 *Industrial Prototype and Reception*

We have designed a working prototype for this problem. Despite the promise that has been made by our industrial partners to provide us real data about historical instances, after more than a year and half we still had not received any data. Therefore, when we have presented our results to conjoint meetings, it was based on instances randomly generated following the procedure mentioned earlier. Therefore, it was not possible to assess the potential gains that our optimization model was able to achieve since we had no schedule to compare our solutions with.

The last time we have presented the work performed on PTP was during a meeting with our industrial partners and their clients, potential future users of our solutions. However, before we had even presented our prototype and the results we had obtained, some clients had clearly stated that they had no interest for this work. Strong sen-

tences were pronounced such as "I don't believe in optimization.". Therefore, after such unenthusiastic reception, even after the showing of our prototype – GUI included – we have decided to tackle another problem in PT centers. This problem is detailed in latter sections.

## 6.3 THE TEN WEEKS AHEAD APPOINTMENT SCHEDULE PROBLEM

When a patient is diagnosed with cancer, he can be treated with Proton Therapy. The PT treatment of a patient corresponds to a sequence of treatment sessions which take place in a PT center. The current problem consists in deciding when to schedule appointments for treatment sessions of a new patient inside a PT center. The global schedule of the PT center in which new appointments have to be inserted is already partially filled with other patient session appointments.

The sequence of treatment sessions is subject to many constraints. Indeed, in order for the treatment to be effective, two successive treatment sessions must be separated by an amount of time whose length lies in a given interval. Such constraints also apply on sets of successive treatment session, e.g., the first and the last session from the set must be separated by an amount of time bounded by a given interval.

To each appointment to be scheduled corresponds a whole treatment session of the patient. The succession of steps the patient must go through during a treatment session corresponds to a specific workflow. All treatment sessions containing the same succession of steps can be characterized as corresponding to the same workflow, even if duration of steps inside of it slightly differ. The steps inside a given workflow are ordered sequentially, i.e., a step cannot begin before the previous one is finished.

Each step from a workflow uses several resources from the PT center in determined quantities. Resources inside a PT center have a maximal capacity. This means that the total consumption of a resource at any point of time should be at most equal to its capacity. For example, a PT center containing 2 scanning rooms cannot scan more than two patients (one patient in each scanning room) simultaneously. As seen in Section 6.1, the resources of a PT center are either unary or cumulative.

The time taken by a given treatment session might depend on the moment at which it is scheduled. For example, a treatment session needing a specific resource and scheduled at a moment at which this resource is used by a large number of other patients might cause de-

lays in the whole workflow of the patient (or in the workflows of other patients). Another example is a session requiring an additional snout change. This occurs when a session is scheduled at a moment when patients have a different tumor type. The limitation of resources imposes that only a few patients can be treated in parallel. A good schedule of patient treatments allows to treat several patients in parallel such that the treatment is as comfortable as possible for both patients and staff members.

There are several objectives for which the schedule of patient appointment of treatment sessions can be optimized. Here is a list of these objectives:

MINIMIZE SESSION TIME
The sessions must be scheduled such that the time needed to complete them is minimal. Indeed, scheduling a session at the wrong moment will result in potential delays occurring during the workflow of the session, due to a resource requirement which could not be fulfilled immediately.

MAXIMIZE ADAPTABILITY
The placement of the session appointments should keep the global PT center schedule as *adaptable* as possible. The adaptability of a given treatment session represents the possibility to move an appointment to another time slot. This allows to keep the schedule to unexpected events such as equipment breaking or staff member sicknesses. As the delay between successive treatment sessions is bounded, some treatment appointment cannot be moved to another time slot without moving other treatment sessions. This kind of sessions are not flexible since their rescheduling imposes to move other appointments. Hence, sessions that have the possibility to be rescheduled without modifying the whole schedule of a patient's remaining sessions are preferred.

MINIMIZE ADDITIONAL SNOUT CHANGES
The number of different snouts needed per day should be minimized. If we reduce the number of different snouts needed each day, we have more chance to produce daily schedules with less snout changes and thus, less wasted time.

MINIMIZE OVERLOADED DAYS
The number of days for which schedules are likely to exceed a given time limit – for example the number of opening hours of the PT center

– should be minimized. When sessions are scheduled either too late or too early, it decreases the comfort of both staff members and patients.

Note that the TWAASP is tightly linked with PTP described in Section 6.1. Indeed, PTP optimizes the schedule of a PT center during a single day. However, we can consider that this optimization can be performed for each day in the PT center calendar. One can see TWAASP as adding patient workflows to days that have previously been optimized with PTP such that a set of constraints are respected, a new set of objectives are optimized along with minimizing the perturbations brought to daily schedules previously optimized with PTP.

## 6.4 A CP MODEL FOR TWAASP

In this section we define our model to solve the TWAASP. The approach chosen is a CP resolution combined with an LNS strategy to propose *reactive* optimization. What we mean by *reactive* solution is that a user will be able to add constraints to a solution that will then be re-optimized to find a new solution respecting the additional constraints.

### 6.4.1 *The Model*

This problem is modeled as a COP$(X, D, C, O)$ where each component is detailed below. While this problem might present similarities with scheduling problems, the model we use here is far more simple. We consider that we have access to all the details of the partial schedule of the PT center i.e., we know what are the details of patient sessions already present in this schedule. In addition to this information, we take as input to our model the sequence of the $n$ sessions to be inserted in the partial schedule and all the details associated to them.

The components of our COP$(X, D, C, O)$ are as follows:

VARIABLES – $X$
To each treatment session $i$ of the patient to insert in the partial schedule corresponds a variable $x_i$ with $1 \leq i \leq n$.

DOMAINS – $D$
Each variable $x_i$ has the following domain: $D_i = [0; \text{horizon}]$ where the value 0 corresponds to the current day in the schedule (i.e. the day at which the request of new session insertion is done) and horizon

corresponds to the last day at which a treatment session can be scheduled (i.e., in our case we assume we cannot see further than 10 weeks ahead in our schedule).

CONSTRAINTS – $C$

First, one has to impose the strict ordering between the successive treatment sessions. Then, the amount of time required between two sessions – either two successive sessions or for a subsequence of successive sessions – is bounded. The amount of time between the session $l$ and the session $k$ is bounded between $\text{minDelay}_{l,k}$ and $\text{maxDelay}_{l,k}$.

OBJECTIVES – $O$

The first objective is to minimize the time that is needed by the additional new treatment sessions. This objective can be expressed with a sum of `Element` constraints. The data from the partial schedule informs us on what are the resources available each day and what sessions using these resources are scheduled on which day. Thanks to this piece of information, we can pre-compute what would be the cost of scheduling a new treatment session $i$ on the period $d$. To compute the time that would be taken by the new treatment session $i$ if scheduled on period $d$, we use a formula taking into account the capacity and usage at period $d$ of the resources used by $i$. This formula could be substituted with a more accurate version, but we will not discuss it any further in this work.

The second objective is to maximize the adaptability of the schedule. We measure the adaptability of the schedule with the delays between sessions. As these delays are bounded, one desires to minimize the number of time this delay is equal to a bound. Indeed, if the delay between session $l$ and session $k$ corresponds to its minimal value $\text{minDelay}_{l,k}$, it will not be possible to advance session $k$ without modifying other sessions in the schedule. The same holds for the maximal value of the delay between two sessions: $\text{maxDelay}_{l,k}$. Hence, we minimize the number of delays between sessions that are either equal to their lower or upper bound. This is done with the help of reified variables.

The third objective is to minimize the additional snout changes required by the new sessions. It is easy to pre-compute whether or not there would be an additional snout change if session $i$ is scheduled at

period $d$. This allows us to model this objective with a sum of `Element` constraints.

Finally, the fourth objective is to minimize the number of overloaded days. Again, it is easy to pre-compute whether or not the period $d$ will be overloaded if session $i$ is scheduled at period $d$. Again, we use a sum of `Element` constraints to model this objective.

These four objectives are aggregated through a weighted sum to ease the resolution process.

Formally, the TWAASP model described above is as follows:

$$\begin{aligned}
\text{minimize} \quad & w_1 \cdot \sum_{i=1}^{n} \texttt{Element}(x_i, \text{estimatedTime}_i) && (6.10)\\
& + \; w_2 \cdot \sum_{l} \sum_{k} \text{minDelayKO}_{l,k} + \text{maxDelayKO}_{l,k} && (6.11)\\
& + \; w_3 \cdot \sum_{i=1}^{n} \texttt{Element}(x_i, \text{additionalSnoutChange}_i) && (6.12)\\
& + \; w_4 \cdot \sum_{i=1}^{n} \texttt{Element}(x_i, \text{periodOverload}_i) && (6.13)\\
\text{such that} \quad & \forall i : \; x_i < x_{i+1} && (6.14)\\
& \forall l, \forall k : \; \text{minDelay}_{l,k} \leq x_k - x_l \leq \text{maxDelay}_{l,k} && (6.15)\\
& \forall l, \forall k : \; \text{minDelayKO}_{l,k} == (x_k - x_l === \text{minDelay}_{l,k}) && (6.16)\\
& \forall l, \forall k : \; \text{maxDelayKO}_{l,k} == (x_k - x_l === \text{maxDelay}_{l,k}) && (6.17)
\end{aligned}$$

where $\text{estimatedTime}_i$ is an array containing for each period $p$ an estimation of the time needed to schedule session $i$, $\text{minDelayKO}_{l,k}$ and $\text{maxDelayKO}_{l,k}$ are reified variables where $==$ is the symbol used to constrain two terms to be equal and $===$ is the reification operator equal to 1 if left and right members are equal and equal to 0 if they are different, $\text{additionalSnoutChange}_i$ is an array containing for each period $p$ the value 1 if an additional snout change is needed by scheduling session $i$ at period $p$ and 0 otherwise, similarly, $\text{periodOverload}_i$ is an array containing for each period $p$ the value 1 if an overload occurs with session $i$ scheduled at $p$ and 0 otherwise.

### 6.4.2 *Simplifications and Potential Improvements of the Model*

Several simplifications of the model have been performed. First of all, two of the four objectives from rely on the pre-computation of respectively $\text{estimatedTime}_i$ and $\text{periodOverload}_i$ arrays. The values put in these arrays have been defined using heuristic functions. However, to be more accurate, for each treatment session $i$ and each period $p$, we could have solved PTP at period $p$ while adding patient session $i$ to the schedule. This would allow to obtain more accurate values in $\text{estimatedTime}_i$ and $\text{periodOverload}_i$ arrays. This would however require to resolve $n \cdot \text{horizon}$ PTPs. Therefore, we have decided to avoid this computationally expensive computation by using heuristic functions.

A second simplification was to consider that patient appointments could be scheduled any time during the day. We have decided to assign a day for each treatment session needed, not an exact hour appointment time. This is done fore several reasons. First of all, as the schedule of the PT center is not yet full for all the days considered, some level of optimization for individual days has to remain. Second, most of the time, patients usually do not have a full schedule yet for all the days where a possible appointment for a treatment session could be set. Finally, most of the time, patients following a PT treatment are in a state of severe weakness. They are therefore monitored in a hospital close to the PT center and are completely dedicated to their treatment, hence they are free at any time for their treatment sessions. As our goal was to demonstrate the feasibility of a reactive optimization framework, we have decided that our periods at which appointments could be set were days. It is easy to extend this prototype to have more granular periods e.g., half days or an hour of a day. Furthermore, as our reactive optimization framework allows additional user constraints, if a patient is not available on some period, it is possible to forbid that any appointment should be scheduled at that given period.

### 6.4.3 *Resolution*

We used CP with LNS to find possible schedules for treatment session appointments. In order to propose a reactive resolution, i.e., to let the opportunity for a user to slightly modify a solution by adding new constraints, we have designed a CP with LNS resolution coupled with a Graphical User Interface (GUI) to represent solutions and interact

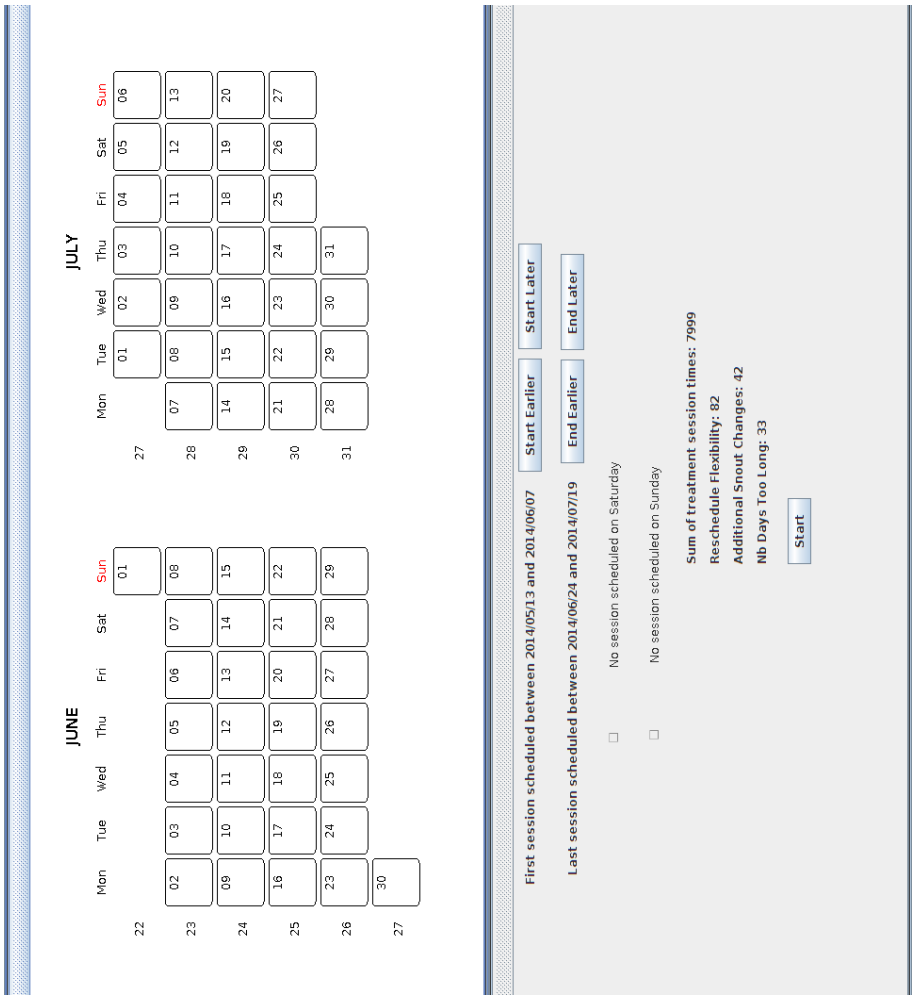



Figure 6.3: GUI used to interact with the model

with our solver. We have used the OscaR open-source solver [Osc12]. The branching and value selection heuristics is very simple since we use a classic binary-first-fail heuristics on the $x_i$ variables.

The CP search quickly finds a feasible solution (in about one second). After a first solution is found, we begin an LNS exploration to improve the quality of the solution. In this LNS approach, the relaxations randomly relax 50% of the variables. This first resolution phase is limited to five seconds as we want to quickly display a feasible solution to the user. This solution is displayed in the GUI shown in Figure 6.3.

Once a solution has been found, the user can add new constraints to modify it. If these constraints still allow to keep partially the last

solution found, an LNS relaxation maintaining this solution part is launched. This allows us to find more rapidly a new feasible solution. However, as new constraints are added/removed, it is not always possible to reach an improved objective value. The upper bound on the objective is therefore relaxed when new constraints are added. In the case where the user has only removed constraints, this upper bound on the objective is maintained.

In Figure 6.3, we observe that the GUI is separated in two distinct parts. The first part at the top of the image is simply a calendar displaying the appointment solution of our model. The user cannot interact with this top part of the GUI. On the other hand, the bottom part of the GUI allows the user to interact with the model. The four first lines of this bottom part defines options to adjust before launching the search and the start button launches the search. The parameters that can be changed in this GUI are:

#### EARLIEST AND LATEST DATES FOR FIRST SESSION

The first line "First session scheduled between $\text{first}_{\min}$ and $\text{first}_{\max}$" is used to add the following constraints on the first appointment:

$$\text{first}_{\min} \leq x_1 \leq \text{first}_{\max}$$

The two buttons "Start Earlier" and "Start Later" allow to respectively decrease and increase $\text{first}_{\min}$ and $\text{first}_{\max}$. The effect of modifying the interval in which the first session can be scheduled will impact the whole sequence of sessions to schedule.

#### EARLIEST AND LATEST DATES FOR LAST SESSION

The second line "Last session scheduled between $\text{last}_{\min}$ and $\text{last}_{\max}$" is used to add the following constraints on the last appointment:

$$\text{last}_{\min} \leq x_n \leq \text{last}_{\max}$$

The two buttons "End Earlier" and "End Later" allow to respectively decrease and increase $\text{last}_{\min}$ and $\text{last}_{\max}$. As for the first session, modifying the interval in which the last session can be scheduled will impact the whole sequence of sessions to schedule.

#### ALLOW SESSIONS ON SATURDAY

The third parameter is a checkbox which adds the following constraints when checked:

$$\forall i \in [1, n] : \quad (x_i \bmod 7) \neq \text{Saturday}$$

where mod is the modulo operator (e.g., $(a \bmod b)$ denotes the remainder when dividing $a$ by $b$) and Saturday represents the integer which represents Saturday in our schedule. Checking this checkbox will thus forbid any session to be scheduled on Saturday.

ALLOW SESSIONS ON SUNDAY

The fourth parameter is a checkbox which adds the following constraints when checked:

$$\forall i \in [1, n] : \quad (x_i \bmod 7) \neq \text{Sunday}$$

where Sunday represents the integer which represents Sunday in our schedule. Checking this checkbox will thus forbid any session to be scheduled on Sunday.

Finally, the last line in the bottom part of the GUI contains a single button: "Start". This button, when pressed, launches a search phase of 5 seconds. During these 5 seconds, the options and the start button are not available and a CP with LNS strategy is performed in the background, modifying the upper visualization part of the GUI. This search begins by adding the new constraints stipulated by the options. Then it iteratively performs LNS relaxations to try to find new solutions improving the current objective value.

Every time the user clicks on the "Start" button, only a given number of relaxations are performed. In order to avoid a specific relaxation to lead to too much exploration, we also impose a dynamic limit on the number of failures allowed per relaxation, similarly to what was proposed in Section 6.2. When this limit is reached, its value is increased if at least one solution was found for this relaxation, otherwise it is decreased. This is done to reduce the loss of time when a good solution has been reached. All those limitations are performed to keep the GUI as reactive as possible.

A lot of other constraints could be implemented to allow more customization of the current solution by the user. However, this model and the GUI associated to it were developed to be a proof of concept. A possible interesting improvement could be to allow the user to adjust the weights of the aggregated objectives.

### 6.4.4 *Industrial Prototype and Reception*

We have designed a working prototype for this problem with a working GUI allowing a user to have some interactions. Similarly to PTP,

we have had no access to any data on which we could have tested our prototype. It has thus been tested on randomly generated instances that tried to represent real situations. On these random instances, we have been able to show the practicability of our reactive optimization approach.

We have presented our prototype to our two main industrial partners: IBA and Palantiris. They were enthusiastic about our prototype, especially because of the GUI. Indeed, should the work that we have performed on TWAASP become part of a commercial software someday, the interaction of the user with our solution is of high interest. While a classical optimization tool allows to obtain high quality schedule, it sometimes simplifies too much the problem by proposing a generic optimization. However, with the GUI and reactive optimization approach we have designed, the user could customize its own model to reach a solution that satisfies his preferences. Our industrial partners have also highlighted the importance of leaving control to the potential users of such applications. Indeed, their point was that leaving the opportunity of changing a proposed schedule would provide a confidence feeling for the potential user. Our industrial partners were also keen on testing the limitations of our approach by over constraining the problem. Despite the fact that at some point our framework was not able to find any feasible solution – because the problem was too constrained – they were impressed with the robustness of our approach since as long as you did not make the problem unsatisfiable, our framework was always able to provide a solution.

This prototype was the last proposed for the MIRROR project. Despite their interest for our prototype, it was not developed as a part of a commercial product. Nevertheless, several ideas that we have introduced here have given some ideas and proven the feasibility of such approaches to our industrial partners and they were pleased with our solution.

## CONCLUSION

In this chapter, we have discussed two problems related to Proton Therapy and the treatment centers in which patients are cured. The first problem was the Proton Therapy Problem in which one attempts to schedule the treatment sessions of patients in a PT center for a given day. This problem is complex and involves several constraints as well as not so trivial optimization objectives. We therefore have proposed a

CP model to tackle it. We have designed to consider a CP resolution strategy for the model we have proposed for PTP. As this model is hard to solve, we have used an LNS strategy to attempt to improve the obtained schedules. This strategy has shown good performance over large instances that are supposed to represent real-world patient mixes and PT center configurations. We have therefore proven the applicability of a CP + LNS approach for large scheduling applications in a complex medical context.

The second problem tackled in this chapter was the Ten Weeks Ahead Appointment Schedule Problem. In this problem, one attempts to place the treatment sessions of a new patient in the partially filled schedule of a PT center. There are various non-trivial optimization objectives to this problem and a CP + LNS approach has been chosen to resolve it. This approach was chosen for its modularity. Indeed, the resolution process has been integrated with a GUI allowing the user to modify a solution by adding/removing constraints to the original model. The results obtained were encouraging and we have thus proven that it was possible to use a CP + LNS approach in a reactive context.

These two problems have finally both been tackled using CP + LNS and the results obtained have shown good performances on both problems. This is yet another proof that CP + LNS is a technique that performs well on problems related to time on real-world problems with large instances.

# 7

# NUCLEAR MEDICINE PATIENT SCHEDULING

*The power of the sun in the palm of my hand.*

—Dr. Otto Octavius, *Spider-Man 2*

*Why do we fall sir? So that we can learn to pick ourselves up.*

—Alfred Pennyworth, *Batman Begins*

*I would like nuclear fusion to become a practical power source. It would provide an inexhaustible supply of energy, without pollution or global warming.*

—Stephen Hawking

*Do* you *know what's on the PET scan?*

—Gregory House, *House, M.D.*

*Reality TV to me is the museum of social decay.*

—Gary Oldman

Nuclear Medicine (NM) is a medical imaging technique in which patients are administered radioactive tracers. As the tracers decay in the human body, they emit photons, which are then captured to generate an image used for diagnostic purposes. The management of doses injected to patient is crucial since it contains radioactive emission hazard. Furthermore, as tracer compounds decay, emissions decrease, requiring larger exposition times in a scanner to obtain image of sufficient quality.

The schedule of daily patients in an NM center is a hard and interesting problem due to the management of radioactive resources. This scheduling problem allows us to define two uncommon scheduling abstractions: continuously degrading resources and interval dependent activity durations. In this chapter, we model the NM scheduling problem as a Constraint Optimization Problem. We have used a resolution strategy combining CP with LNS. We also detail two propagation procedures to deal with continuously degrading resources and interval dependent activity durations.

In Section 7.1, we describe the Nuclear Medicine Problem and its context. Then, Section 7.2 defines a model for NMP. This model includes continuously degrading resources and interval dependent activity durations whose propagation procedures are explained in Section 7.3. Finally, we assess the results obtained by our model on real world sized instances in Section 7.4.

### RELATED PUBLICATIONS

[DD14] Cyrille Dejemeppe and Yves Deville. "Continuously Degrading Resource and Interval Dependent Activity Durations in Nuclear Medicine Patient Scheduling." In: *Integration of AI and OR Techniques in Constraint Programming*. Cork, Ireland: Springer International Publishing, 2014, pp. 284–292.

## 7.1 THE NUCLEAR MEDICINE PROBLEM

Nuclear Medicine (NM) is a clinical practice in which patients are administered nuclear tracers in order to provide diagnostic information for a wide range of diseases. This technique is notably widely used to diagnose and follow the treatment of cancer tumors. The nuclear

tracers injected to patients are mixes of radioactive compounds. The emissions, also referred to as radioactive activity, decrease over time as the compounds decay. As defined in [FIo2], the activity of a radioactive tracer decreases with time according to the following law of decay:

$$\mathrm{Rad}(t) = \mathrm{Rad}_0 \times \mathrm{e}^{\frac{-t\ln(2)}{t_{0.5}}} \tag{7.1}$$

where $\mathrm{Rad}_0$ is the initial activity of the decaying substance and $t_{0.5}$ is its half-life time. The decay of a nuclear tracer through time is represented in Figure 7.1.

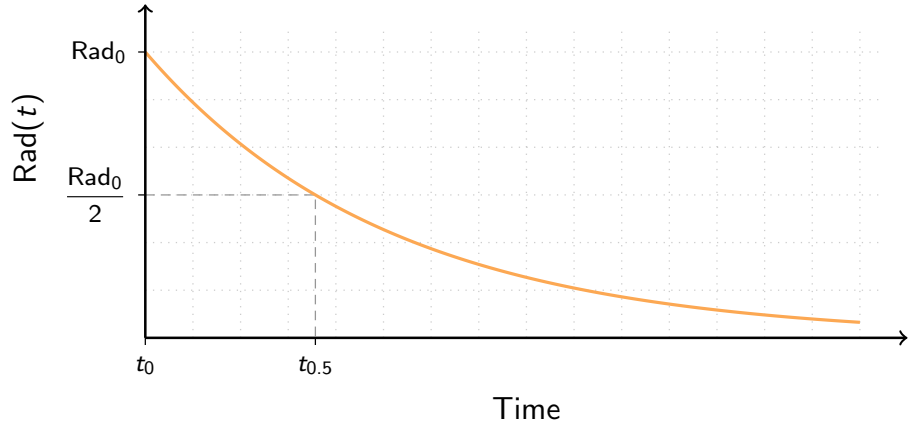



Figure 7.1: Evolution of the radiation level of a nuclear tracer with time. The value $t_0$ represents the starting time of the decay. At the half-life time of the tracer, the radioactivity level has decreased by half its original level: $\mathrm{Rad}(t_{0.5}) = \frac{1}{2}\mathrm{Rad}_0$.

During decay, the radioactive compounds in the tracer emit gamma rays or high-energy photons. The energy levels of these emissions are such that a significant amount of energy can exit the body without being scattered or attenuated. External gamma-ray sensors allow capturing these emissions, and computers are then able to recreate an image from them. The quantity of emissions required by the scanner to get an image of sufficient quality depends on the area of the body that has to be scanned. Hence, the quantity of radioactive tracer injected to a patient depends on the body area to scan. Furthermore, the weight of a patient is inversely proportional to the quantity of emissions leaving its body. This means that heavier patients will be injected larger doses of tracer.

As stated in [CSP12], NM has several advantages over other medical imaging techniques. Most other imaging techniques are either more

intrusive i.e., they bring a larger discomfort to patients or they produce less accurate images. The precision and the quality of the images obtained with NM makes it a technique widely used for medical imaging.

The Nuclear Medicine Problem (NMP) consists in scheduling the workflow of patients inside an NM center such that several objectives are optimized.

### 7.1.1 *NM Treatment Center*

There are more and more Nuclear Medicine centers installed around the world. While these centers differ on many points, most of them share a common structure. The structure that is described here is inspired by the NM center in Mont-Godinne, Belgium. There can be many variants of this structure and most of the centers are variants of a base model. However, most of the time, they only differ by the number of rooms they contain and thus the number of patients that can be treated simultaneously.

The main rooms of NM centers are the following ones:

COLD WAITING ROOM
When the patient arrives to the NM center, he has to wait until his turn comes. According to perturbations and delays, some patients might begin their treatment earlier – if they are in the waiting room in advance – or later than the predicted appointment time. However, we will not consider perturbations and delays i.e., this work does not perform *robust* scheduling. Instead, we expect that the schedule that we will produce will be followed without any unexpected event occurring.

INJECTION ROOM
The injection room is the room in which doses are administered to patients. The injection is performed by a robotic arm for two specific reasons. First, it avoids the staff to manipulate too much nuclear substances. Indeed, each member of the staff is equipped with a dosimeter that records the level of radioactivity that he has been exposed to. The amount recorded by this dosimeter cannot exceed a given limit. This limit is set to the maximal exposure that is known not to be harmful for the human body. Should a human be exposed to radiations above this limit, his health could be put in danger. The second reason for which a robotic arm is used for injection is that it avoids

human mistakes that could lead to spills of the tracer. Not only the tracer substance is expensive, but it also could be a hazard for the patient if some drops of the substance were spilled directly on his skin.

#### HOT WAITING ROOM

Once the patient has been injected, he waits in another waiting room in order to allow his body to incorporate the tracer. The *hot* waiting room is completely separated from the *cold* waiting room. Patients that have been injected must be separated from other patients since they start emitting radioactive emissions as their tracer decays. The duration of this waiting time has to last at least a minimal amount of time, otherwise the human body will not have fully incorporated the nuclear tracer. On the other hand, the waiting time cannot be too long: this would mean that the radioactive tracer would have decayed for too long, leading to its radioactive activity being too low to provide satisfactory images.

#### SCANNER ROOM

Once his body has fully incorporated the tracer, the patient goes into a scanning room in which the image is captured. There can be several different scanning rooms which differ in their scanner equipment. The amount of time needed by the scanner equipment to capture the image directly depends on the amount of time the patient has been waiting after having been administered the radioactive tracer. To adjust image quality, the exposition to the scanner is sometimes increased to capture even more emissions. We however will not consider in this work that the duration of the scanning of a patient can be adjusted on the fly to change the quality of the image. Instead, we consider here that the scanning time depends only on the duration of the waiting time of the patient in the hot waiting room. As stated in [Sch+03], the acquisition time can be expressed as a linear function of the waiting time as follows:

$$d^{\text{acq}} = \alpha + \beta \cdot d^{\text{wait}} \tag{7.2}$$

where $d^{\text{acq}}$ and $d^{\text{wait}}$ are the respective durations of the acquisition and waiting time, $\alpha$ and $\beta$ are positive constants depending on the quantity and type of tracer administered to the patient. Figure 7.2 shows an example of how the acquisition time depends on the waiting time. These scanner rooms exist in different instances, corresponding to the type of image desired. Some of these scanner room in-

stances are duplicated to allow parallel scans of patients requiring the same image type.

LABORATORY

The nuclear substances are stored and prepared in this room. At the beginning of the day, nuclear components are delivered to the NM center. These nuclear components are very expensive and only few specialized institutions can produce them. These nuclear components are stocked in multiple containers. As long as a container remains sealed, the decay of nuclear components remains negligible. However, as soon as the seal of a container is removed, the nuclear components start to decay. Hence, to avoid *wasting* nuclear emissions (and also for safety reasons), containers are unsealed only when required. The nuclear components from a container are mixed with saline solution to become a nuclear tracer. The nuclear tracer doses are prepared inside the laboratory.

Note that in opposition to PT centers from Chapter 6, the corridors are not of high importance here. Indeed, congestion does not need to be taken into account in corridors as either patient walk by themselves or they are transported in wheeling chairs. This means that several patients can walk by easily when they cross each other in corridors. Furthermore, it does not happen that patients have to wait in corridors, they are directly going from one room to the other one, according to their respective workflow.

Once the seal of a nuclear container has been broken, the radioactive level of its content decreases as shown in Figure 7.1. The captors for scanning the patients capture emissions coming out of their body. In order to capture an image of a given quality, emissions coming out of a patient's body should correspond to a given level. Hence patients should be injected a given *dose* of radioactivity. The tracer injected to patients comes from a mix of nuclear compounds from a nuclear container unsealed for some time. Hence, the tracer they are injected with has already decayed a bit and thus will have a lower radioactive activity. To match a given dose, the quantity of tracer injected to a patient will vary according to its state of decay. This means that the quantity of nuclear tracer a patient is injected with is inversely proportional to the radioactivity of a tracer. Formally, for a patient $i$, the quantity of the tracer of type deg injected at time $t$, $\mathrm{Q}_i^{\mathrm{deg}}(t)$, is defined as follows:

$$\mathrm{Q}_i^{\mathrm{deg}}(t) = \gamma_i \cdot \mathrm{e}^{\frac{t \ln(2)}{t_{0.5}^{\mathrm{deg}}}} \tag{7.3}$$

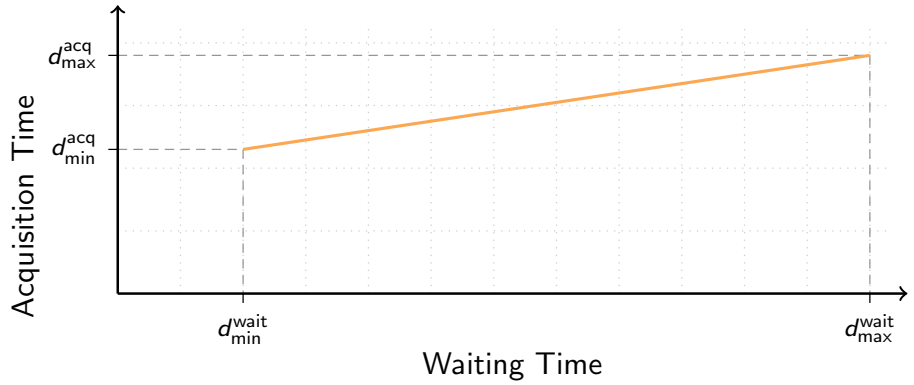



Figure 7.2: Time required for image acquisition in function of the length of the waiting time after injection. Here, $d^{wait}_{min}$ and $d^{wait}_{max}$ correspond respectively to the min and max duration of the waiting time. Similarly, $d^{acq}_{min}$ and $d^{acq}_{max}$ are the the min and max duration of the acquisition time. In this example, we have the following values from Equation (7.2): $\alpha = 1.0$ and $\beta = 1.15$.

where $t^{deg}_{0.5}$ is the half-life time of the nuclear components in the tracer deg.

This quantity is limited by a maximal value. Indeed, after some time, not only should the patient be injected with unrealistically large amounts of tracer, but the radioactivity level of the tracer would be too low. Hence, the content of a nuclear component whose seal has been broken is not exploited after an amount of time $t_{max}$.

An example of the evolution of the quantity of tracer injected to a patient varying with time (the time since the seal of the nuclear container has been broken) is shown in Figure 7.3.

### 7.1.2 *Patient Workflow*

Patients coming to an NM center are treated following the steps of a workflow. For a given patient, the steps of his workflow might differ in terms of duration, but the sequence of steps he goes through is defined by his workflow. In NM centers different workflows follow the same sequence of steps but they however differ by the pieces of equipment needed at each step. Here are the successive steps followed by a patient treated in an NM center:

1. The patient arrives in the NM center and waits for the beginning of its appointment in the cold waiting room.

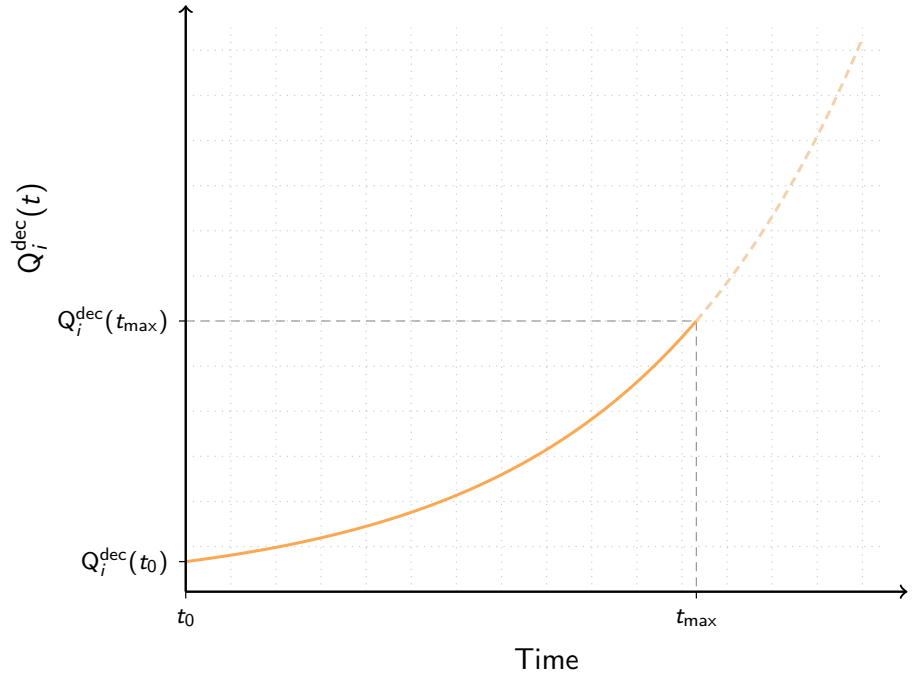



Figure 7.3: Evolution of the quantity of tracer injected to a patient with time. The value $t_0$ represents the time at which the seal of the nuclear container has been broken.

2. The patient is administered a nuclear tracer in the injection room.
3. The patient waits in the hot waiting room to allow his body to incorporate the tracer.
4. The patient is scanned in the scanning room corresponding to the type of image desired.
5. The patient leaves the NM center.

By the time a patient leaves an NM center, the emissions coming out of his body are too low to be harmful to other people. The results of the imaging process are sent with analysis to the physician in charge of the patient treatment. The physician then contacts the patient and, depending on the diagnosis, prescribes a treatment.

### 7.1.3 *NMP Scheduling Objectives*

The Nuclear Medicine Problem (NMP) consists in scheduling the steps of patient workflows inside an NM center. This scheduling is performed such that two objectives are optimized. Here are these two objectives:

PATIENT THROUGHPUT
: The patient throughput represents the number of patients that can be treated on a single day. If more patients can be scanned in a day, then more patients could be scanned in an NM center. One objective is thus to maximize the patient throughput.

TRACER CONSUMPTION
: The quantity of tracer injected depends both on parameters specific to the patient and on the time at which the patient is injected. As nuclear tracers contain very expensive materials, one objective is to minimize the total amount of tracer used by the schedule.

One of the goals of NMP is to combine these objectives. We would then provide a set of solutions which quality can be quantified according to both criteria described above. This set of solutions would be an approximation of the Pareto set of our problem. A human user would then select one of the schedules proposed in the Pareto set. This means that these objectives are not aggregated and we will consider bi-objective optimization in our resolution.

## 7.2 THE MODEL

We propose in this section a scheduling model for NMP. This model introduces two uncommon concepts. The tracer is modeled as a continuously degrading resource, i.e. a resource starting with an initial level that is only decreased, never increased. The level of a continuously degrading resource can never be negative. The second uncommon abstraction introduced by our model is the interval dependent activity. An interval dependent activity is an activity whose duration is a function of the length of the idle time between two activities. The main components of our scheduling model will be described here: activities, resources, constraints and objectives.

ACTIVITIES
: Every workflow of a patient is represented as a job and every step

of the workflow is represented as an activity in that job. The steps inside a patient's workflow cannot be interrupted. The activities are thus non-preemptive. Each job contains only two activities: injection and acquisition. The various waiting times are not modeled as activities. The duration of the injection activity is known and fixed. The duration of the acquisition activity is bounded and depends on the amount of time between the end of the injection and the start of the acquisition.

RESOURCES

The two main resources considered in this work are the injection room and the various acquisition (scanner) rooms. As most center have a single injection room, it is modeled with an unary resource. The acquisition rooms depend on the configuration of the center. Some centers contain a specific acquisition room in several instances and other ones on a single instance only. Consequently, unique acquisition rooms are modeled with a unary resource while those present in multiple instances are modeled with cumulative resources. Note that the rooms represented with cumulative resources should be modeled with alternative resources to avoid that a patient has to switch from acquisition room in the middle of the acquisition process.

CONSTRAINTS

The first constraints of our problem define an order between injection and acquisition activities from the same job. Furthermore, as the body of the patient needs a minimal amount of time to integrate the tracer, a minimal delay is imposed between successive injection and acquisition. Similarly, the waiting time of a patient between his injection and acquisition activities cannot exceed a maximal amount of time.

Another constraint is needed to determine the duration of the acquisition activity. Indeed, the acquisition time depends on the amount of time between injection and acquisition. The duration of the acquisition activity is defined as a linear function of the waiting time after injection as shown in Equation (7.2).

Finally, we have to express the constraints on the continuously degrading resources (tracers). We have decided not to model the tracer as a resource. Instead, we keep a counter of the amount of nuclear tracer that is consumed through time by injection activities. To do so, we consider that injection activities consume an increasing amount of tracer over time as expressed in Equation (7.3). We assume that the

amount of a resource needed by an activity is consumed at its starting time in an atomic way.

OBJECTIVES

This problem considers two objective functions, as detailed earlier. We represent the maximization of the throughput with a makespan minimization. Indeed, if a given number of patients can be treated in a shorter amount of time, then there could be additional patient scans added on the same day. The second objective is the minimization of the total consumption of tracer resources. Note that, as said earlier, these two objectives will not be aggregated but considered in a multi-objective context.

Formally, the model described above is as follows:

$$
\begin{aligned}
\text{minimize} \quad & \begin{cases} \texttt{makespan} \\ \displaystyle\sum_{\text{deg}} \sum_{i} w_{\text{deg}} \cdot \mathrm{Q}_i^{\text{deg}}(s_i^{\text{inj}}) \end{cases} \\
\text{such that} \quad & \forall i : s_i^{\text{acq}} - e_i^{\text{inj}} \geq \text{minDelay} \\
& \forall i : s_i^{\text{acq}} - e_i^{\text{inj}} \leq \text{maxDelay} \\
& d_i^{\text{acq}} = \alpha_i + \beta_i \cdot (s_i^{\text{acq}} - e_i^{\text{inj}}) \\
& \forall ur : \texttt{UnaryResource}\,(\mathcal{A}, U_{ur}) \\
& \forall cr : \texttt{CumulativeResource}\,(\mathcal{A}, U_{cr}, C_{cr}) \\
& \forall \text{deg} : \textstyle\sum_i \mathrm{Q}_i^{\text{deg}}(s_i^{\text{inj}}) \leq \text{initialCapacity}(R_{\text{deg}})
\end{aligned}
$$

where $R_{\text{deg}}$ represents a continuously degrading resource, $w_{\text{deg}}$ is a positive weight determining the importance of the consumption of $R_{\text{deg}}$, $i$ represents a patient/job, minDelay and maxDelay represent respectively the minimal and maximal waiting time of a patient between his injection and acquisition activities, $\alpha_i$ and $\beta_i$ are positive integer constants, $\mathcal{A}$ is the set of all activities, $ur$ denotes a unary resource and $U_{ur}$ is a vector defining for each activity whether or not it uses $ur$, $cr$ is a cumulative resource, $U_{cr}$ is a vector defining for each activity the usage of $cr$ and $C_{cr}$ is the capacity of $cr$ and initialCapacity$(R_{\text{deg}})$ is the initial available amount of the continuously degrading resource $R_{\text{deg}}$.

The second objective minimizes the weighted total consumptions of tracers. The first and second constraints impose respectively a minimal and maximal waiting time between injection and acquisition activities.

The third constraint is the interval dependent activity constraint linking the duration of the acquisition to the duration of the waiting time. The fourth and fifth constraints model the use of treatment rooms (injection room and the various acquisition rooms). Finally, the last constraint limits the total usage of the continuously degrading resources to never exceed their initial capacities.

While this model is simple and concise, there are no dedicated propagation procedure for the two uncommon abstractions introduced: continuously degrading resources and interval dependent activities. The next section explains how propagation can be performed for those.

## 7.3 DEDICATED PROPAGATION PROCEDURES

In this section, we describe how to perform propagation for the continuously degrading resource constraint and the interval dependent activity duration constraint. Both these propagation procedures rely on the use of *view-based propagator derivation*. We first recall what is view-based propagator derivation, then we instantiate this technique to the continuously degrading resource and the interval dependent activity duration constraints.

### 7.3.1 *View-Based Propagator Derivation*

The *view-based propagator derivation* technique [SSo8] allows to apply propagators on *transformations* of domains. A view is represented by two functions, a transformation function $\phi$ and its inverse $\phi^{-1}$. Given a propagator $p$, a view is represented by two functions $\phi$ and $\phi^{-1}$ that are composed with $p$ to obtain the desired propagator $\phi^{-1} \circ p \circ \phi$. Starting from a domain $D$, these two functions allow to apply a propagator $p$ on a transformation of the domain $D_\phi$. The filtering of $p$ is applied on the transformed domain $D_\phi$ and its filtered version $D'_\phi$ is obtained. The inverse transformation $\phi^{-1}$ is then applied on $D'_\phi$ to obtain the filtered original domain $D'$. Concretely, the propagation is performed by applying the following steps:

1. Apply $\phi$ to the original domain $D$ to obtain the transformed domain $D_\phi$.
2. Apply propagator $p$ on $D_\phi$ and obtain its filtered version $D'_\phi$.
3. Apply $\phi^{-1}$ on $D'_\phi$ to obtain the original filtered domain $D'$.

These view functions can be combined and allow to chain transformations. If we consider two transformation functions $\phi_1$ and $\phi_2$ and their respective inverse $\phi_1^{-1}$ and $\phi_2^{-1}$, to apply the propagator $p$, one can use the following procedure to perform filtering on the original domain: $\phi_1^{-1} \circ \phi_2^{-1} \circ p \circ \phi_2 \circ \phi_1$.

Let us consider a small example. Let us consider a variable $x$ and its current domain $D(x) = \{1, 2, 3, 4\}$. The following constraint is applied to the domain:

$$3 \cdot x \leq 8$$

We can apply view-based propagator derivation by considering the following transform function $\phi$ and its inverse $\phi^{-1}$:

$$\phi(v) = 3 \cdot v$$
$$\phi^{-1}(v) = \tfrac{1}{3} \cdot v$$

The propagator $p$ removes all the values of a domain that are larger than 8:

$$p(D) = \{v \mid v \in D \wedge v \leq 8\}$$

With these two functions and this propagator, the propagation procedure is able to obtain the filtered domain $D' = \{1, 2\}$ following the steps detailed in Figure 7.4.

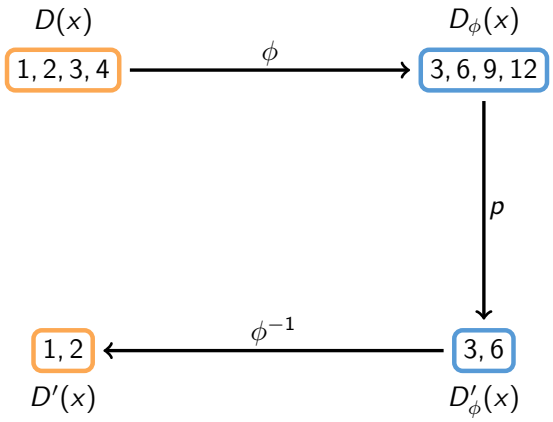



Figure 7.4: Example of application for view-based propagation derivation from [SS08].

### 7.3.2 *Continuously Degrading Resource*

The continuously degrading resource constraint seem complex at first since it involves using an exponential function on a variable. However, with the help of view-based propagator derivation, we will be able to achieve Bound Consistency (BC) for this constraint. A BC propagator ensures that the constraint is verified for the domain bounds (min and max values of the domain). As a reminder, here is the constraint for which we have to design a propagation procedure:

$$\sum_i \gamma_i \cdot e^{\frac{s_i^{\text{inj}} \ln(2)}{t_{0.5}^{\text{deg}}}} \leq \text{initialCapacity}(R_{\text{deg}}) \tag{7.4}$$

We propose to define a transformation function $\phi_i$ for each patient $i$ and each continuously degrading resource $R_{\text{deg}}$ as follows:

$$\phi_i(v) = \gamma_i \cdot e^{\frac{v \ln(2)}{t_{0.5}^{\text{deg}}}}$$

Its inverse version is as follows:

$$\phi_i^{-1}(v) = \frac{t_{0.5}^{\text{deg}}}{\ln(2)} \cdot \ln\left(\frac{v}{\gamma_i}\right)$$

As values returned by the $\phi$ function are real values and our CP variables only accept integer values, we consider the domain of $s_i^{\text{inj}}$ mapped by $\phi$ is a discrete domain in which each value corresponds to a single value in the domain of $s_i^{\text{inj}}$. This means that any real value obtained by applying either $\phi$ or $\phi^{-1}$ will be rounded up (either to the upper or lower integer value depending whether we are dealing respectively with the result of ). The definitions of $\phi$ and $\phi^{-1}$ allow us to use a classic linear sum constraint propagator as proposed in [Apto3].

We can see that with the help of views, we were able to obtain a BC propagation procedure without large implementation efforts.

### 7.3.3 *Interval Dependent Activity Durations*

The interval dependent activity duration constraint involves three variables and two constants. As a reminder, the constraint we are concerned with is the following one:

$$d_i^{\text{acq}} = \alpha_i + \beta_i \cdot (s_i^{\text{acq}} - e_i^{\text{inj}}) \tag{7.5}$$

The first step we can perform to simplify the constraint is to create an auxiliary variable $d_i^{\text{wait}}$ that represents the waiting time of patient $i$ between its injection and acquisition activities. This auxiliary variable is constrained as follows:

$$d_i^{\text{wait}} = s_i^{\text{acq}} - e_i^{\text{inj}}$$

With this auxiliary variable, we are able to rewrite the constraint as follows:

$$d_i^{\text{acq}} = \alpha_i + \beta_i \cdot d_i^{\text{wait}}$$

We propose to apply two successive transformation functions on the right part of this constraint. Let us define the following two transformation functions:

$$\chi_i(v) = \beta_i \cdot v$$
$$\psi_i(v) = \alpha_i + v$$

Their inverse versions are as follows:

$$\chi_i^{-1}(v) = \frac{v}{\beta_i}$$
$$\psi_i^{-1}(v) = v - \alpha_i$$

Considering the propagator $p$ for the equality constraint, we are able to use view-based propagator derivation applying the following operations:

$$\chi^{-1} \circ \psi^{-1} \circ p \circ \psi \circ \chi$$

Let us consider a small example with these transformation functions. In this example, we have $\alpha_1 = 6$ and $\beta_i = 2$. The transformation functions defined earlier are as follows:

$$\chi(v) = 2 \cdot v$$
$$\psi(v) = v + 6$$
$$\chi^{-1}(v) = \tfrac{1}{2} \cdot v$$
$$\psi^{-1}(v) = v - 6$$

The respective domains of the acquisition duration $d_i^{\text{acq}}$ and the waiting time duration $d_i^{\text{wait}}$ variables are $D(d_i^{\text{acq}}) = \{16, 18\}$ and $D(d_i^{\text{wait}}) = \{4, 5, 6, 7\}$. The GAC equality propagator $p$ removes all the values of

the transformed waiting time duration domain, $D_{\chi \circ \psi}(d_i^{\text{wait}})$ that are not equal to any value of the acquisition duration domain $D(d_i^{\text{acq}})$:

$$p(D_1) = \left\{ v \mid v \in D_{\chi \circ \psi}(d_i^{\text{wait}}) \wedge v \in D(d_i^{\text{acq}}) \right\}$$

With these two transformation functions, their inverse and this propagator, the propagation procedure is able to obtain the filtered domain $D'(d_i^{\text{wait}}) = \{5, 6\}$ following the steps detailed in Figure 7.5.

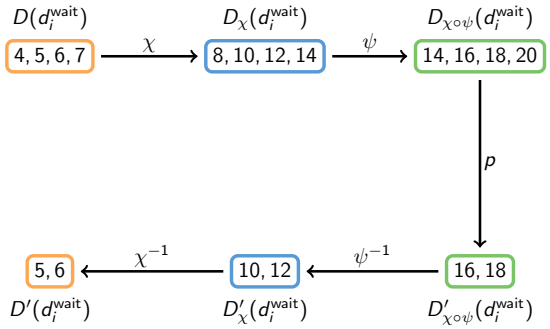



Figure 7.5: Example of application of the view-based propagator derivation for interval dependent activity durations.

Similarly to continuously degrading resources, we were able to obtain a propagation procedure without large implementation efforts with the help of views.

### 7.3.4 *Simplifications and Potential Improvements of the Model*

Several simplifications of the model have been performed. First of all, acquisition rooms present in several instances have been modeled with cumulative resources. This could however cause some patient to switch acquisition room in the middle of its acquisition process as cumulative resources do not avoid activities overlapping. To overcome this problem, alternative resources should be used instead to assign each acquisition activity to a single scanning room. We have not done this in our model for a very simple reason: the two NM centers we have visited did not have acquisition rooms in several instances. These NM centers had several acquisition rooms, but the scanning equipment they contained was different from one room to another. Hence, depending on the type of scan desired, only a single scanning room was available. We propose in our model that several instances of the same type of

acquisition room exist only because we have heard that it might be possible in other NM centers.

Another simplification of this model is that the staff members are not represented. According to our visit, there were enough members (much more than needed) to ensure the full function of the centers. Staff members are either associated to a treatment room (injection or acquisition) or they are there to monitor patients and make junctions between the rooms. As the number of parallel activities requiring staff members never exceeds the number of available staff members, they are not modeled. This is done to avoid a complication of the model and an overhead in the resolution time by running propagators that will never fail nor prune. The staff members should however be taken into account if some objective representing the comfort of staff members were to be added to the current model.

Finally, a simplification we have made in our model is that we do not take into account maintenance times that are sometimes needed. As explained earlier, the injection activity is performed by a robotic arm that might need maintenance. We have decided not to include these maintenance in our model as they are usually planned every week. To model a day in which a maintenance operation occurs, one only has to fix an activity using the injection room at the time at which the maintenance is planned. The duration of this maintenance activity is fixed and thus does not bring any uncertainty to our model.

## 7.4 RESULTS

To give an overview of the complex nature of NMP, we propose to solve four different versions of the problem. Each version increases the level of complexity of the previous version. These problems are solved using a CP strategy combined with LNS. The branching heuristic used for search is a binary first fail on the start variables $s_i$. Similarly to the previous chapter, the choice of LNS relaxation and the associated parameters was performed from experience. We have run experiments on a set of possible well known LNS relaxations while varying their parameters, the seed of the random generator, etc. One relaxation has obtained the best results on most instances, except a few ones where it was the second best by a small margin. Hence, in order to ease the benchmarking process, we have selected this single relaxation strategy. It imposes that start variables $s_i$ are set to the values of the last solution except for 10% of them that are randomly chosen.

Here are the descriptions of the four models that have been used in our experiments:

MODEL $V_1$
This model is a relaxation of NMP in which neither continuously decreasing resources nor the interval dependent activity durations are considered. This means that durations are fixed for all activities and that we do not limit the usage of nuclear tracers. Hence, the only objective considered is the minimization of the makespan, this objective is denoted MK in our results. The model $V_1$ is thus a classic model for a Cumulative Job-Shop Problem.

MODEL $V_2$
The second model $V_2$ is an extension of the $V_1$ model where the interval dependent activity duration constraints have been added. Hence, the durations of the acquisition activities of patients are not fixed anymore and we add the constraint stated in Equation (7.5). Again, we only consider minimization of the makespan (MK).

MODEL $V_3$
The third model $V_3$ is an extension of the $V_2$ model where the continuously degrading resource constraints, expressed in Equation (7.4), have been added. As this allows us to model the quantity of tracer used easily, the objective function has been modified. The model does not consider makespan minimization but it focuses on minimizing the quantity of tracer used (this objective is denoted TQ in our results).

MODEL $V_4$
The fourth model $V_4$ considers NMP as a bi-objective problem. It is an extension of model $V_3$ where both objectives MK and TQ are minimized. Hence, this model aims at finding a set of non-dominated solutions instead of a single one. To solve this problem we use the constraint introduced in [SH13]. This constraint allows to easily maintain a set of non-dominated solutions in a multi-objective optimization context in CP. Note that a possibility to perform this multi-objective optimization could have been to use the Variable Objective LNS introduced by Schaus in [Sch13a].

For each version of our problem, a time limit of three minutes is imposed and the best values found for both objectives (makespan and quantity of tracer consumed) are reported in Table 7.1. All experiments

were conducted with the OscaR open-source solver [Osc12]. The instances considered are lists from 10 to 50 patients obtained by a biased random generator we designed. The durations of patient activities, the resource capacities, and the decay parameters are generated using realistic values. However, typical NM centers with the considered configurations treat at most 25 patients per day and larger instances are considered only to test the limits of the model.

For the first model $V_1$, we can observe that the quantity of tracer used (TQ) increases dramatically with the number of patients and with the makespan (MK). This is due to the exponential nature of the quantity of tracer required with time, as stated in Equation (7.3).

When comparing the results from Table 7.1 for models $V_1$ and $V_2$, we can see that the makespan and quantity of resource used are higher for $V_2$ than for $V_1$. This can be explained by two main reasons. First, the solutions for model $V_1$ are not solutions of model $V_2$. Indeed, $V_2$ adds a relation linking the waiting time of patients with the duration of imagery acquisition durations. This relation could not be respected in a solution for model $V_1$. Second, as $V_2$ does not fix the duration of the acquisition activities, the search space is larger for $V_2$ than for $V_1$. As both problems $V_1$ and $V_2$ have ran under the same conditions and with the same branching heuristics, it is normal that the resolution of $V_2$ fails to obtain solutions as good as the ones obtained by solving $V_1$.

As expected, when comparing results for models $V_2$ and $V_3$ in Table 7.1, we observe that the quantity of tracer used is on average lower for $V_3$ than for $V_2$ as opposed to the makespan which is higher. This was expected as they minimize respectively makespan and tracer quantity while not considering the other objective. To obtain solutions which are tradeoff between the two objective functions, it is interesting to consider a bi-objective version of our problem as proposed in $V_4$.

The results from Table 7.1 for $V_4$ are the average best solutions obtained for both objectives. We can observe the reported average of the best solutions obtained are between the best and the worst values found for $V_2$ and $V_3$ for the makespan and the quantity of tracer used.

In Figure 7.6, we report the Pareto front obtained solving $V_4$ as well as the best solutions obtained resolving $V_2$ and $V_3$ on instances with 20 and 40 patients. We observe that some solutions obtained by $V_4$ are dominated by the best solutions obtained by $V_2$ and $V_3$. On the other hand, some other solutions are not dominated by the best solution for $V_2$ nor the best for $V_3$. This can be explained by the fact that the resolu-

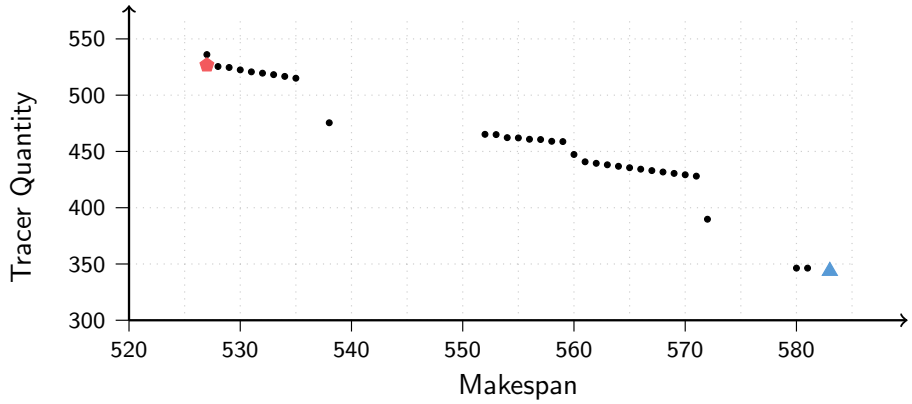
| Model | 10 Patients | | 20 Patients | | 30 Patients | | 40 Patients | | 50 Patients | |
|---|---|---|---|---|---|---|---|---|---|---|
| | MK | TQ | MK | TQ | MK | TQ | MK | TQ | MK | TQ |
| Model $V_1$ | 251 | 9.97 | 446 | 40.44 | 650 | 129.9 | 867 | 516 | 1,048 | 1,268 |
| Model $V_2$ | 253 | 11.49 | 486 | 51.16 | 737 | 242.6 | 994 | 1,065 | 1,211 | 2,770 |
| Model $V_3$ | 291 | 9.17 | 530 | 39.04 | 779 | 164.7 | 1,029 | 671 | 1,245 | 1,862 |
| Model $V_4$ | *266* | *9.42* | *495* | *38.32* | *751* | *182.2* | *1,011* | *757* | *1,234* | *1,885* |

Table 7.1: Average objective values for the four different models on different sizes of instances (10 instances per size; 50 instances in total). MK is the makespan and TQ is the total quantity of tracer used. For models $V_1$, $V_2$ and $V3$, the values reported are the average values for the instance size considered. For problem $V_4$, values reported are the averages of the best values found for each objective for the instance size considered.

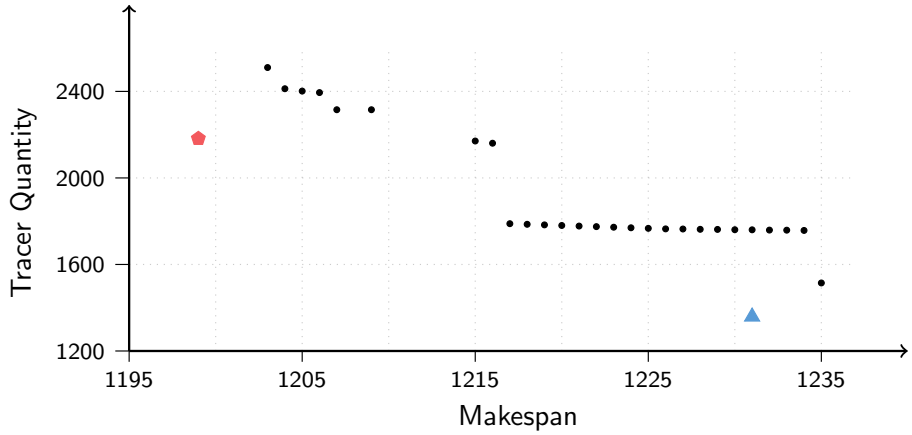
tions of the different models explore different parts of the search space. This effect can be further reinforced with the randomness brought by the LNS strategy. Another interesting detail to observe is that the fronts propose a range of well spread solutions. The front is *dense* in the sense that there are several solutions between extreme solutions (those with the smallest objective value for a single objective). Furthermore, these fronts always contain solutions close to the best solutions from $V_1$ and $V_2$. Finally, the many solutions between the extreme points are well spread between those. As such, the problem version $V_4$ is well suited to obtain a set of tradeoffs between the two objectives considered.

### 7.4.1 *Industrial Prototype and Reception*

We have designed a working prototype with a GUI for this problem. This prototype was well received by our industrial partners. However, due to the lack of data and hence the impossibility to assess the potential gains of such approach, it has not yet been used in a hospital environment. Nevertheless, some components of the prototype we have designed were of high interest for the industrial partners. Unfortunately for us, the aspects they were most interested in were not especially related to optimization. Two main outcomes of what we have done here could be used on a given fixed schedule. First, they were able to obtain a decent approximation of the quantity of tracer used on a single day. This can be done using our activity representation in correlation to our




(a) Instance with 20 patients




(b) Instance with 40 patients

Figure 7.6: Comparison of Pareto front solutions obtained with resolution of model $V_4$ and the best solutions obtained by solving models $V_2$ and $V_3$. The red pentagons are the best solutions obtained on model $V_2$, the blue triangles are the best solutions obtained on model $V_3$ and the black circles are the points of the Pareto front obtained with model $V_4$.

continuously decreasing resource representation. Second, the fact that we were able to show them that sometimes injecting a patient later could reduce the amount of time he spent in an acquisition activity. Indeed, with our model representing the acquisition duration proportional to the waiting time between injection and acquisition, they were able to determine the best time at which the injection should be per-

formed to obtain the desired acquisition times and desired waiting times.

## CONCLUSION

In this chapter, we have described the Nuclear Medicine Problem. We have modeled it as a scheduling problem. To deal with some characteristics of the problem, we have introduced two uncommon scheduling abstractions: continuously decreasing resources and interval dependent activity durations. These two scheduling abstractions were easy to model but could not be linked to any efficient existing propagation procedure. Indeed, these constraints may seem complex at first since they link several varaibles together and they include non-linear function e.g. an exponential function. With the help of view-based propagator derivation, we have proposed a simple propagation procedure making use of existing propagators. One of the main advantages of these propagation procedures using views is that they can be implemented with minimal efforts.

Finally, we have proposed an efficient method to solve NMP using a CP + LNS approach. We have proposed four different models embedding an increasing number of constraints, allowing finer modelization, closer to reality. Each model can be solved according to the desired objective function. The last version has even allowed us to perform bi-objective optimization to obtain a set of solutions which are tradeoffs between both objectives.

The proposed modeling and search techniques are generic and could be used for other cumulative scheduling problems with specific constraints. The only requirement is that these specific constraints combine existing constraints (i.e., with an existing propagator) on new variables which are defined as functions of variables of the initial problem (e.g., start and end activity variables). Thanks to the use of views, propagators of these constraints can be applied. Our approach allows a large range of cumulative scheduling problems with specific additional constraints.

Part IV

# REDUCING ENERGY COSTS IN PRODUCTION PLANNING

# 8

# PAPER PRODUCTION PLANNING

*Do I really look like a guy with a plan?*

—The joker, *The Dark Knight*

*Be prepared!*

—Scar, *The Lion King*

*A goal without a plan is just a wish.*

—Antoine de Saint-Exupéry

*Failing to plan is planning to fail.*

—Alan Lakein

*Adventure is just bad planning.*

—Roald Amundsen

The share of renewable energy production, such as wind or solar power is growing fast in several countries of the EU [WB06]. While the production of nuclear and fossil energy tends to be stable, renewable energy production is highly dependent of both climatic conditions and the time of the day considered. Renewable resources add a huge variability on energy offer and demand, and thus on the price of electricity. As an example, Figure 8.1 shows the historical electricity prices in Europe on March 3rd, 2014. In this example, the electricity prices fluctuate with a multiplicative factor higher than 3.5. Performing activities requiring more energy when electricity price is low represents both an economic and ecologic advantage (the energy produced is not "wasted").

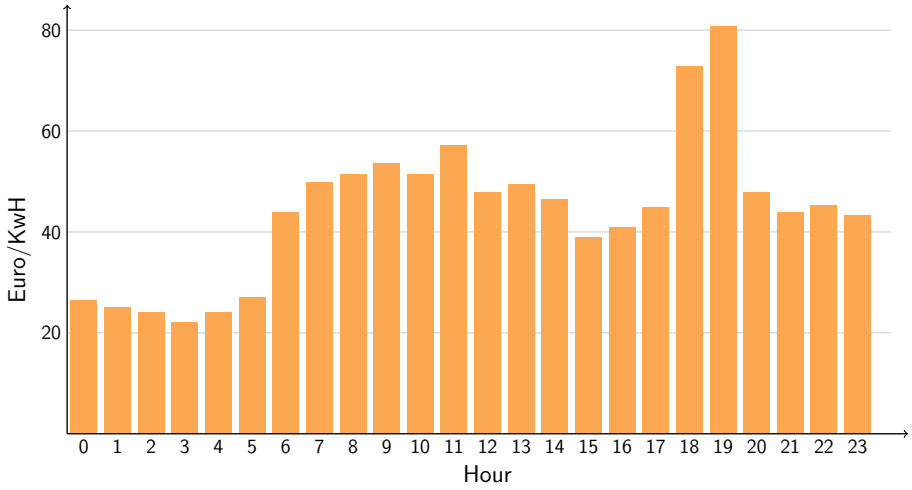



Figure 8.1: Historical evolution of electricity prices on the EU market on March 3rd, 2014.

In [SH11], Simonis and Hadzic propose a cumulative constraint that links the energy consumption of activities with evolving electricity prices. We believe this kind of energy-aware optimization will become increasingly present in the industries with order-driven production planning that can be easily split into different steps. It generally offers enough flexibility to reduce the energy costs by scheduling activities requiring more energy when the electricity price is lower.

This chapter addresses the problem of energy-efficient scheduling in consumer tissue production planning. Consumer tissue production planning offers several levers of flexibility, allowing to drastically reduce the energy costs for a given set of orders. Indeed, the paper ma-

chine receiving paper pulp as input and producing paper rolls consumes an amount of energy that depends on the tissue properties (quality, density of fibers, thickness, etc).

This chapter therefore describes a model attempting to schedule the production of paper rolls requiring more energy when the electricity price forecasts are lower. Two CP models representing different parts of the production process are described in this chapter. These two models are solved sequentially in order to reach a global reduction of energy costs in the whole production process.

In Section 8.1, we describe the consumer tissue manufacturing problem. Then, in Section 8.2, we propose a modelization of this industrial problem using two CP models. Finally, Section 8.3 explains the results obtained on real historical data with our model.

### RELATED PUBLICATIONS

[Dej+16] Cyrille Dejemeppe, Olivier Devolder, Victor Lecomte, and Pierre Schaus. "Forward-Checking Filtering for Nested Cardinality Constraints: Application to an Energy Cost-Aware Production Planning Problem for Tissue Manufacturing." In: *Integration of AI and OR Techniques in Constraint Programming*. Banff, Canada: Springer International Publishing, 2016.

## 8.1 PAPER PRODUCTION PLANNING

An important industrial site in Belgium manufactures hygienic paper (toilet paper and facial tissues are examples of refined paper they produce). Paper rolls are produced before being converted into different products (e.g. toilet papers or kitchen rolls). The production is a two step process: paper roll production, then conversion of paper rolls into final products. In Figure 8.2, we give a schematic overview of the different steps in the production of paper on the industrial site considered.

Depending on the type of paper produced, the energy consumption required by the machines can vary. This actually depends on several parameters for the type of paper: the mix between long and short wood fiber, the thickness of the paper, etc. The energy consumption can vary by up to 15% depending on the type of paper roll produced. Therefore

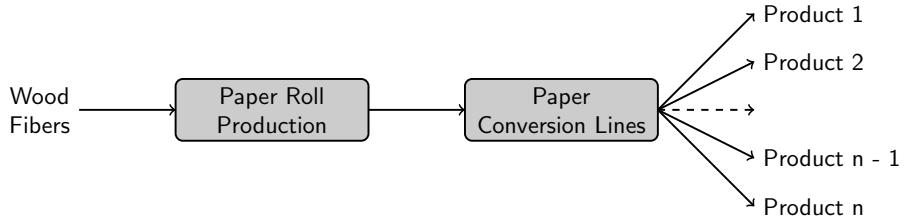



Figure 8.2: Production steps in paper industry.

the company is looking for the less expensive production planning given the electricity price forecasts.

The potential savings depend on the flexibility of the production site. For example, a factory continuously producing the same product does not have much potential to reduce its energy bill. On the contrary, a manufacturer producing many different products on a production line, each requiring a significantly different amount of energy has probably more flexibility to reduce its energy bill.

The paper conversion lines directly convert paper rolls into final products. These conversion lines are however not dependent on the type of paper they process. The only lever for flexibility here would be to produce orders in advance and store them in a stock. However, the size of the stock is limited, leaving us with a tiny potential energy gain. Furthermore, the energy consumption of conversion lines is significantly less energy-intensive than the two other steps of the production process. Optimizing the production plan of the converting lines based on electricity costs is therefore not considered. We will thus focus on the paper roll production part of the production line in this thesis.

The paper roll production can be separated in two main successive steps: paper pulp production and transformation of paper pulp into paper rolls. We will briefly detail these two steps.

### 8.1.1 *Pulp Preparation*

The paper pulp production consists in the mixing of wood fibers with water in the desired proportion. There are two types of wood fibers: short and long. The proportion of short and long fibers in the paper pulp will give certain properties to the paper produced and the exact mix is driven by the type of paper desired as a product. The mix of long fibers, short fibers and water is performed in two different pulpers.

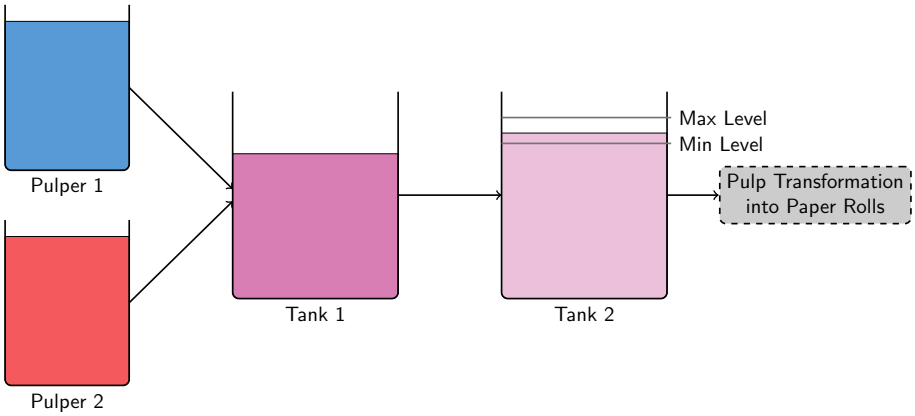



Figure 8.3: Paper pulp production.

Once the paper pulp obtained has been mixed long enough, it is spilled in one large tank, that himself is spilled in another large tank. This last tank is used to aliment the paper machine that produces paper rolls starting from the paper pulp. As the paper machine runs continuously, there is a constant debit of paper pulp from the second tank.

Typically, at the considered industrial site, the pulpers have a countenance of 35% of the large tanks. The only part of this process needing a relevant amount of energy is the mixing of fibers and water inside the pulpers. The available flexibility on this part of the process is to use the tanks and the pulpers as buffers.

For example, if we know that the electricity prices will jump soon, we could fill the two tanks and make the two pulpers process their batch without emptying them. Then, during the period where the electricity price is high, as a lot of paper pulp will already be ready in the tanks and the pulpers, the pulpers will not be needed, reducing the amount of energy consumed at that period. Figure 8.3 illustrates the paper pulp preparation process in a schematic way.

Note that on Figure 8.3, the colors represent different mixes. This means that the content of a pulper, when spilled into tank 1, will alter the mix in this tank (and the same holds when a portion of the content of tank 1 is spilled into tank 2). Furthermore, we can notice that the second tank is constrained by a minimum and a maximum level. This means that the content of the second tank must always be between the min and the max level imposed.

### 8.1.2 *Paper Machine*

The paper machine transforms paper pulp into paper rolls. This consists in a continuous process where the paper pulp is spread out on a conveyor belt passing through several presses and in front of several heating devices or ventilation systems in order to dry the pulp and obtain a sheet of paper that will then be rolled up to form paper rolls. The number of rolls of a given type of paper is determined by an order book. This order book imposes that there must be at least $n$ rolls of a given type available at time period $p$. The rolls can be produced in advance and stored in a stock. At period $p$ when the $n$ rolls of the considered paper type are required by the order book, $n$ rolls of this paper type are removed from the stock.

As this process is continuous, the biggest factor that can impact the consumption of electricity is the kind of paper that has to go through the process. Indeed, depending on the type of paper pulp on the conveyor belt, the speed of the belt, the temperature of the heater, the speed of the ventilation systems and several other parameters will vary. The flexibility of this part of the production resides in the permutation of paper types according to electricity prices.

For a given type of paper, the conveyor belt and the other components of the production line have to be calibrated. As such, to avoid too many calibrations, when a paper type is produced, it has to be produced for a minimal period of time before another paper of a different type can be produced. Furthermore, as calibration is an exhaustive and expensive process (because the quality of the paper cannot be ensured during a transition between two different paper types), productions plan in which the same paper type is produced for large periods are preferred (i.e., there are less switches between different paper types).

The calibration time and loss of paper quality both depend on the two paper types between which they occur. As such, some transitions between successive paper types produced are more desirable than others. A transition cost can thus be associated for every transition (i.e. every pair of paper types that will be produced successively). There are also some transitions that are forbidden as they could harm the different machines involved in the process.

## 8.2 MODELING THE PRODUCTION PROCESS

In this section, we describe a scheduling model to represent the production of paper rolls. First, we have decided to separate the two parts of the production process (paper pulp production and transformation of paper pulp into paper rolls) into two different models that we will link together. Indeed, considering the whole production line in a single model would greatly complicate the model. Furthermore, the search space would be much larger on a single unified model and it would result in much larger solving times.

The first model will be a model for the transformation of paper pulp into paper rolls in the paper machine. Indeed, considering a set of demands (i.e., demands from the order book of a given amount of paper rolls of a given type before a given due date), it is natural to first consider the order in which the rolls of each paper type appearing in the demands will be produced.

Once the order of types of paper rolls is determined, this can be considered as the output flow of the model of the paper pulp production. This is convenient because this will allow us to have the mix of different pulps defined a priori and this makes the paper pulp model much easier. Furthermore, the lever of optimization is higher in this model as the flexibility of the corresponding part of the production line is much higher than the other one. Indeed, once the output flow of the paper pulp production is fixed, our only lever of optimization is the filling of the pulpers and tanks. As, for the site we have visited, the combination of the capacity of tanks and pulpers (if they are not emptied) allow us to store a bit more than 7 batches (one batch represents one full pulper), we cannot hope to optimize for a long period of time should the electricity prices remain high for a long period. Furthermore, the electric consumption of the pulpers is about 20 times lower than for the refinement of the pulp into paper rolls process.

For all those reasons combined, we have chosen to first solve the paper machine model (transformation of the pulp into paper rolls). From the solution obtained on the paper machine model, we then solve the paper pulp production model. A resolution of these two models in the opposite order would greatly complicate the models (especially to cope with the mix of different pulps) and would probably result into schedules of lower quality. In Section 8.2.1 we propose a model for the paper machine while Section 8.2.2 describes a model for the paper pulp production.

### 8.2.1 *Paper Machine Model*

The paper machine model should include the following constraints:

- For every demand of paper rolls of a given type at a specified due date, a larger or equivalent amount of paper rolls of the same type has to be produced before the respective due date.
- When a roll of a given paper type is produced, it has to be produced for a minimal amount of time before any other paper type can be produced.

As for the objectives, we have the following quantities to optimize:

- The total energy cost of the production planning has to be minimized.
- The cost (and thus also the number) of transitions between different successive paper types has to be minimized. When a transition between two paper types is forbidden, an infinite cost is associated to it. This allows to keep a small and concise model.

We propose a CP model for these constraints and these objectives. We have chosen a discrete representation of the process as the price of electricity at every hour (forecasts) is one of the input in our model. We associate one decision variable for each hour on the horizon on which the production planning has to be optimized. We represent our variables as follows: $X = \{x_1, \ldots, x_n\}$ where $x_1$ is the variable associated to the first hour of the production plan and $x_n$ to the last one (the horizon). The order book is directly translated in full hours to fit to the model. The values these variables can take represent the type of paper produced at the corresponding period. The variable $x_i$ will thus take as value the paper type of the paper rolls to be produced at the hour $i$ of the production plan. This allows easy linking of every production of paper type to the price of the electricity at the corresponding hour.

The first constraint to impose on these variables is to respect the demands of paper rolls. We can do that easily by imposing a Global Cardinality Constraint (GCC) [Rég96] for every deadline that we have in the order book. This constraint will enforce that there are at least-/at most $m$ variables (hours) taking the value $v$ (paper type) on the specified sub-array.

We have analyzed the theoretical pruning provided by this constraint decomposition and we have noticed that it could be improved.

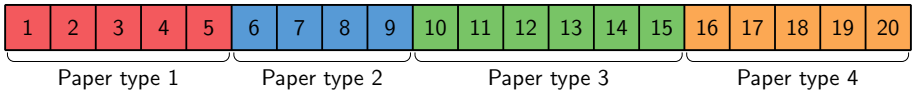



Figure 8.4: Schedule in which the regular constraint is respected.

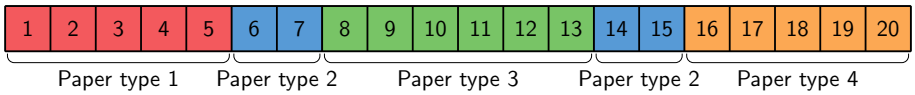



Figure 8.5: Schedule in which the regular constraint is not respected.

Indeed, the various GCCs will not be able to communicate between each other event if their scopes are nested ranges. The Nested GCC from [ZP07] allows to overcome this limitation. Zanarini and Pesant have proposed a GAC propagator for this constraint and we have described a FWC propagation procedure in Chapter 4. As our model will be solved using the help of a LNS approach, we have assumed that the heavy GAC propagator for the Nested GCC from [ZP07] would be too time-consuming for this application. Indeed, its worst time complexity is very large. Furthermore, obtaining an efficient implementation of the GAC propagator from Zanarini and Pesant is rather hard. This is the motivation that lead us to develop a dedicated FWC propagator for the Nested GCC as described in Chapter 4.

The second constraint to impose is that there are at least $k$ successive periods of every paper type produced. This means that, in our representation, the same value has to appear in a succession of at least $k$ variables. To impose this constraint, we have used a Regular constraint [QW06] where the automaton provided as input imposes a chain of $k$ successive same values before reaching an accepting state. If we represent the values (paper types) by colors and variables (hours) as small rectangles, we can represent a feasible schedule as a succession of colored rectangles. In Figure 8.4, we see a schedule in which the regular constraint is respected for $k = 4$ (i.e. there is no succession of rectangles of the same color whose length is inferior to 4). On the other hand, Figure 8.5 shows an infeasible schedule for the same set of paper types produced since there are two successions of only 2 periods where the paper type is blue

The two objectives, minimization of electricity costs and minimization of transition costs between paper types, are aggregated in a sum that is minimized. The first objective is represented as a sum of vari-

ables obtained with the Element constraints. The Element constraint is used to obtain the energy consumption associated to the value (i.e., paper type of the rolls produced) of the variable $x_i$ at hour $i$. This allows us to multiply the electricity price of a period with the amount of electricity consumed at that period by a given paper type production. This implies that we have an array whose elements are the prices of electricity (indices of the arrays correspond to periods) and another array containing the consumption of electricity for producing a given paper type. The second objective is also expressed with a sum of variables defined with Element constraints. However, in this case, it is expressed on a matrix instead of a one-dimension array. Indeed, we have to represent the pairs of paper types in a matrix $T$ where an entry $t_{a,b}$ represents the cost of the transition from the production of paper type $a$ to the production of paper type $b$. This matrix must ensure that the diagonal (representing the transition from any paper type to the same paper type) is null (since we do not want to penalize production plans where there are no real transitions, i.e., no calibration will be needed).

Formally, the paper machine model is as follows:

$$\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{n} \text{price}_i \cdot \texttt{Element}\left(x_i, C\right) + \sum_{i=1}^{n-1} \texttt{Element2D}\left(x_i, x_{i+1}, T\right) \\
\text{such that} \quad & \texttt{Nested_GCC}\left(\left[X^1, \ldots, X^p\right], \left[l^1, \ldots, l^p\right], \left[u^1, \ldots, u^p\right]\right) \\
& \texttt{Regular}\left(\texttt{StretchAutomaton}\left(X, k\right)\right)
\end{aligned}$$

where the first sum of the objective represents the total electricity cost and the second sum represents the transition costs. The bounds $l^r$ and $u^r$ in the signature of the Nested GCC (see Chapter 4 for a definition of this signature) are vectors representing respectively the minimal (imposed by the order book) and maximal (imposed by the stock limitations) number of paper rolls to produce in the specified range of variables $X^r$. $\texttt{StretchAutomaton}\left(X, k\right)$ is an automaton accepting only chains containing the same value at least $k$ successive times. Note that in this particular model, we have not weighted the two objectives, but this could be done in practice to determine their relative importance.

### 8.2.2 *Pulp Production Model*

Once an optimized production plan has been obtained for the paper machine, the output flow of the paper pulp production is fixed. Therefore, we do not have to consider pulp mixes or different types of

fibers. This makes this model easier. Furthermore, we can consider the two tanks as a single tank whose capacity is the sum of the capacity of the two tanks. This allows us to have a lighter and more concise model. Hence, our model only has to express the information of when the batches are processed into the pulpers and when the pulpers are spilled into the tank.

We propose to use a scheduling model in which the processing of a batch in a pulper is represented as an activity. Similarly, the spilling of a pulper into the tank is also represented by a activity. This allows us to easily represent the model with precedences between activities, expressing the ordering in which batches have to be processed. This model includes the following constraints:

- Every batch occurring in a pulper has to be performed from the beginning to the end without being interrupted. As such, there can be only a single batch processed at any point of time for a given pulper.
- The capacity of the tanks cannot be exceeded. Both tanks have a lower and an upper capacity. Therefore, the minimal and maximal capacities of the big virtual tank representing their content are obtained by summing respectively the minimal and maximal capacities of both tanks.
- The spilling of a batch into the tank can be performed only when the batch has been completely processed in the pulper.
- Every batch must be spilled in the tank before its due date. The due date of every batch has been deduced from the output flow obtained by the solution to the paper machine model.

The objective is to minimize the cost of the production plan according to the electricity prices of the period at which a batch is processed.

We propose a CP scheduling model for these constraints and this objective. We have chosen a time step of 5 minutes since it corresponds to the duration of the spilling of a batch from a pulper to the first tank. We associate an activity to each of the following events:

- Every batch $i$ mixed in a pulper is represented with a activity $A_i^{\text{mix}}$. These activities all have the same duration: 20 minutes (4 time steps).

- The spilling of the batch $i$ from a pulper to the first tank is represented with an activity $A_i^{\text{spill}}$. These activities all have the same duration: 5 minutes (1 time step).
- At every time step $t$, we associate an activity $A_t^{\text{cons}}$ that corresponds to the consumption of a fixed amount of pulp from the second tank by the paper machine.

To ensure that no two batches can be processed simultaneously on the same pulper we use a cumulative resource constraint [AB93] with capacity 2 (since there are two pulpers). This constraint will impose that there cannot be more than two simultaneous batches in pulpers at any point in time. There exists a trivial algorithm to find back to which pulper (pulper 1 or pulper 2) each batch is associated. This greedy algorithm associates every activity to the first free resource in lexicographic order. To restrict the amount of pulp inside the aggregated tank, we use a reservoir resource [SC95]. This implies that at the end of every activity $A_i^{\text{spill}}$ (spilling of a batch into the tank), the level of the reservoir resource is increased by the quantity of pulp in the batch. Similarly, every activity $A_t^{\text{cons}}$ (consumption of pulp by the paper machine) is a consumer activity: the level of the reservoir resource is decreased by a given quantity when it begins. To impose that the spilling of a batch can only be performed when the pulper has finished to mix it, we use precedence constraints. For every batch, we impose that the pulper activity precedes the spilling task: $A_i^{\text{mix}} \ll A_i^{\text{spill}}$. Finally, to ensure each batch is performed before a due date, we restrict the initial domain of variables such that the variable corresponding to the end of the spilling of a batch is set to be at most equal to the due date corresponding to the batch.

The objective can be represented, as for the first model, by a sum of variables obtained with element constraints. To ease this process, we pre-compute for every point of time $t$ what would be the electricity cost of a pulper activity $A_i^{\text{mix}}$ starting in $t$. This result is stored in an array pulpElecCost.

Formally, the pulp production model is as follows:

$$
\begin{aligned}
\text{minimize} \quad & \sum_{i=1}^{\mathrm{n}} \texttt{Element}\left(s_i^{\text{mix}}, \text{pulpElecCost}\right) \\
\text{such that} \quad & \forall i \in [1,n]: \quad A_i^{\text{mix}} \ll A_i^{\text{spill}} \\
& \texttt{ReservoirResource}\left(\begin{array}{l} [A_1^{\text{spill}}, \ldots, A_n^{\text{spill}}], \\ \left[A_1^{\text{cons}}, \ldots, A_{\text{horizon}}^{\text{cons}}\right], \\ L_{\text{min}}, \\ L_{\text{max}} \end{array}\right) \\
& \texttt{CumulativeResource}\left(\left[A_1^{\text{mix}}, \ldots, A_n^{\text{mix}}\right], 2\right)
\end{aligned}
$$

where $s_i^{\text{mix}}$ is the start variable of activity $A_i^{\text{mix}}$. In the signature of `ReservoirResource`, the first array $[A_1^{\text{spill}}, \ldots, A_n^{\text{spill}}]$ represents the producers and the second array the consumers $\left[A_1^{\text{cons}}, \ldots, A_{\text{horizon}}^{\text{cons}}\right]$. The values $L_{\text{min}}$ and $L_{\text{max}}$ represent respectively the min and max level of the aggregated tank (obtained by summing the min and max capacities of the individual tanks). Finally, in the signature of the cumulative resource constraint, `CumulativeResource`, the value 2 is the capacity of the cumulative resource.

### 8.2.3 *Simplifications and Potential Improvements of the Models*

Several simplifications of the model have been performed. First of all, we have separated the problem into two disjoint models: paper pulp production and paper machine models. These models are solved in sequence: first the paper machine model fixes the *output* of the paper pulp model, then the paper pulp model is solved. A single model would result in finer optimization and could lead to better solutions. However, as mentioned earlier, our main objective is to reduce the energy costs that are proportional to the energy consumption of the processes to schedule. The paper pulp model contains activities with almost negligible consumption in comparison to the paper machine. Furthermore, the flexibility of the paper pulp production model is small: only the tanks and the pulper can be kept full to avoid running a pulper while prices are high. This is one of the reasons for which it is acceptable to split the problem into two distinct models that will be solved sequentially.

Another simplification is that the pulp model does not model the mix of various paper pulp that can occur in the two tanks. Indeed, we have simplified our model by aggregating the two tanks in a single reservoir resource and even then, we do not bother with the mix of paper pulps. Taking the mixing of various paper pulps would have resulted in a much harder modeling, and much less concise. We have chosen to make this simplification for two main reasons. First of all, mix of various paper pulps only happen during very small periods in the whole schedule. Then, the output of the paper pulp model is fixed. Hence, knowing the mix that is used by the paper machine at any point of time, it is possible to adjust the fiber mix in both pulpers to obtain the desired mix in the end.

Another simplification of these models is the granularity of the time we have used here. As prices are provided hour by hour, we have chosen to use hour time steps. However, to be a bit more accurate, one could use half-hours time steps. According to the visits our industrials partners have made on site, it is not useful to use a smaller time step than half an hour. Our model could be easily adapted to consider half hours time steps instead of hours.

Finally, we have not considered in this model any kind of unexpected event. It however happens that some piece of the paper machine must be put under maintenance or that a pulper or stocking tank has to be cleaned. These events are however unexpected events not planned in advance. Therefore, it would require to perform *robust scheduling*[KDVkoo] that would imply a completely different resolution strategy.

## 8.3 RESULTS

Our complete modelization of the problem is performed through two models solved successively. We had access to partial historical data. This historical data contains the amount and type of paper rolls produced from paper pulp over a couple of years. The data obtained gives us access to the electricity consumption of the various parts of the process. Additionally, we have also been granted access to the historical electricity prices during the same period. Our intention was therefore to test both models and compare the optimized results obtained on this site. Unfortunately, we had to lower our expectations on this matter due to three main reasons:

1. The data was not available for the pulp mixing part of the process. Hence, we have not been able to test the associated model on real data.

2. The historical data does not always respect all the constraints that we were asked to implement. This means that if we consider the historical production plan applied during a given period, some constraints are not respected. The particular constraint that was violated was the one imposing that there should be at least $k$ successive periods producing the same paper type. We have decided to limit our experimentations on periods that were respecting all the constraints.

3. The data is partial. During some periods, no production has been recorded at all. We have no idea of whether the industrial site was actually shut down during these periods or if some maintenance was performed or if the recording devices stopped working. Hence, we have decided to only consider periods where all (or almost all) data was available.

For the reasons mentioned earlier, we have been able to test only the paper machine model. The rest of the experiments described here are performed only on this paper machine model.

### 8.3.1 *The Historical Instances*

Combining the data coming from the industrial site and the historical European electricity prices over the same period, we were able to produce instances as follows:

1. Randomly select two dates separated by a specified amount of days. This defines the time window $tw$ representing the instance.

2. Collect over $tw$ the historical type of paper roll produced every hour.

3. Collect over $tw$ the historical European electricity prices every hour.

4. Collect over $tw$, for every paper type $i$ the contiguous periods at which $i$ is produced. Let $[t_1, t_2]$ be such an interval where product type $i$ is produced continuously. A deadline is imposed

to produce at least $t_2 - t_1 + 1$ items for date $t_2 + \delta$. This $\delta$ value depends on the number of days considered and varies between 24 and 72 hours.

The shifting of deadlines by $\delta$ gives some flexibility to the model for optimization. As we don't have the historical stock constraints, we only impose over the whole time window $tw$ to produce the exact same type and numbers of rolls. We have generated 4 sets of 10 instances for planning of respectively 4, 6, 8 and 11 days (96, 144, 192 and 264 time periods).

### 8.3.2 *Evaluation of Nested GCC Propagation*

As the paper machine model has lead us to develop a FWC propagation procedure for the Nested GCC from Chapter 4, we have decided to evaluate our new procedure on this particular model. To do so, we compare several models including different propagation procedures. All these models are based on the one described in Section 8.2.1 and only differ by the propagation procedure for the Nested GCC. We propose to compare the same three FWC propagation procedures described in Chapter 4 and recalled here:

1. Multiple GCCs FWC propagator decomposition with the original bounds ($\phi_{\text{GCCFWCs}}$); there is one GCC FWC propagator for each range constrained by the original bounds.

2. Multiple GCCs FWC propagator decomposition with strengthened bounds from Section 4.3.1 ($\phi_{\text{PrecompGCCFWCs}}$); there is one GCC FWC propagator for each range constrained by strengthened bounds.

3. The Nested GCC FWC propagator with $\mathcal{O}(\log(p))$ time complexity from Section 4.3.2 ($\phi_{\text{NestedGCCFWC}}$). This version also uses the stronger bounds after the pre-computation step.

These models and propagation procedures have all been implemented in the open-source solver OscaR [Osc12]. Similarly to what has been done in Chapter 4, we have used the replay strategy approach from Van Cauwelaert [CLS15] (described in Chapter 3). The baseline was $\phi_{\text{GCCFWCs}}$ and the sequence of nodes was generated using a binary first fail strategy for 300 seconds per instance.

In Figure 8.6, we can see that approaches using the pre-computation step have a much smaller number of backtracks. Note that, as expected, once the new bounds have been computed, both $\phi_{\text{PrecompGCCFWCs}}$ and $\phi_{\text{NestedGCCFWC}}$ offer the same pruning. We can also see that for about 35% of the instances, these propagators were able to almost completely cut the search tree explored by $\phi_{\text{GCCFWCs}}$. Finally, we can observe that there are only a bit less than 15% of the instances for which the propagators using pre-computed bounds are not able to achieve more pruning than $\phi_{\text{GCCFWCs}}$. All these observations on a model using real world instances give even more credit to the conclusions that we have made in Chapter 4.

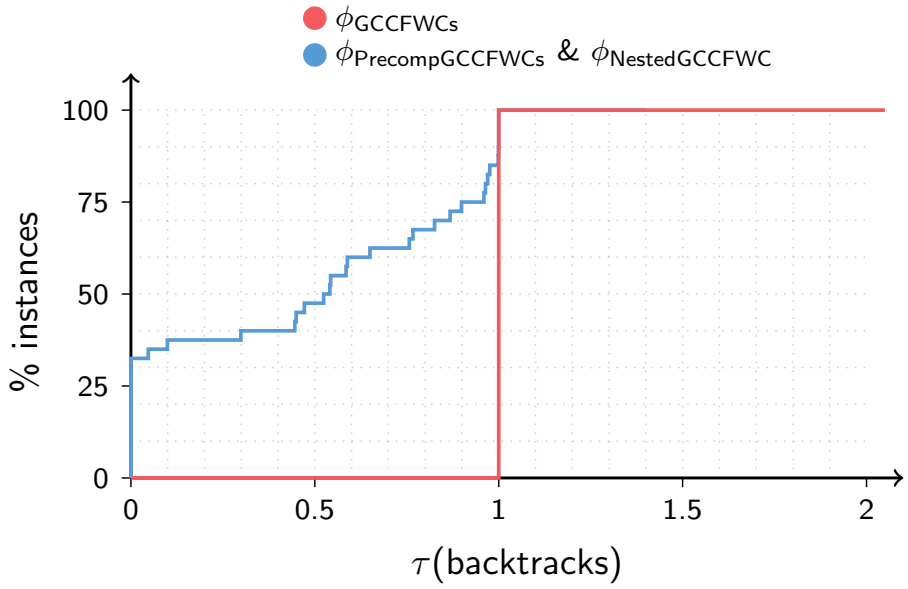



Figure 8.6: Performance profiles of the three models for the number of backtracks.

In Figure 8.7, we can see the profiles of resolution times for the different propagators. We can observe that both $\phi_{\text{PrecompGCCFWCs}}$ and $\phi_{\text{NestedGCCFWC}}$ are faster than $\phi_{\text{GCCFWCs}}$ for about 90% of the instances. The reason is the stronger filtering that is induced by the bounds-strengthening procedure. The 10% of instances for which both these variants are slower than $\phi_{\text{GCCFWCs}}$ are those on which they offer no additional pruning; and even in this case, they are at worst less than 1.5 time slower than $\phi_{\text{GCCFWCs}}$. We can see however that resolution times are similar for $\phi_{\text{PrecompGCCFWCs}}$ and $\phi_{\text{NestedGCCFWC}}$. Again, these

results on real world instances validate the results we have obtained on random instances in Chapter 4.

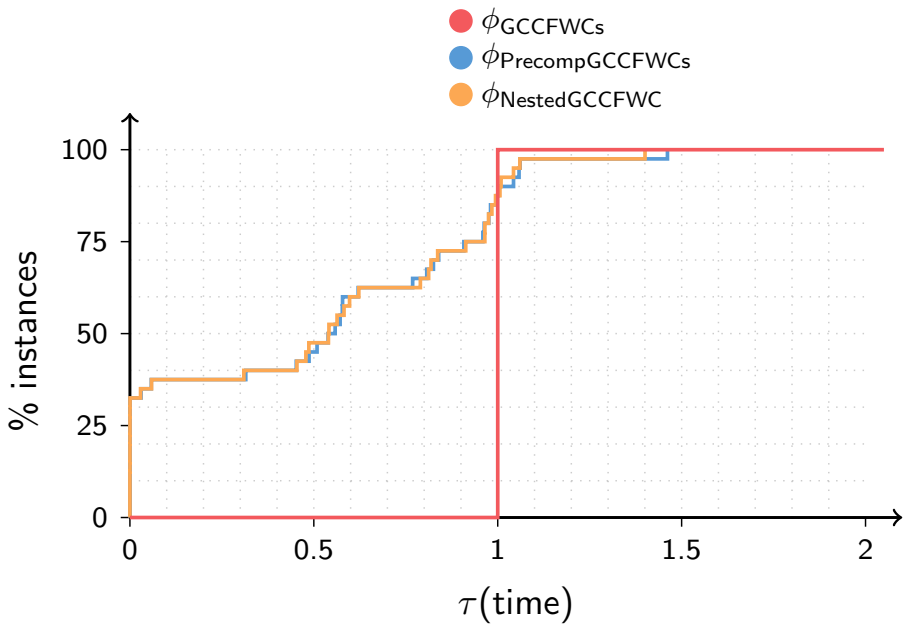



Figure 8.7: Performance profiles of the three models for the replay time on a sequence of nodes as explained in Chapter 3.

*Energy Consumption Minimization with LNS*

This section aims at showing the potential improvement brought by our model over historical production plans. An LNS is used with our CP model from Section 8.2.1 over the historical data and we compare the reductions in terms of cost. The search strategy used is Conflict Ordering Search [GHPS15]. The LNS setting is the following: at each iteration, we select 80% of the possible values (i.e., paper types). Variables associated to these 80% of values are then relaxed. This is done to relax the production plan except some *blocks of production* over some specific paper types. The search is stopped if one of these two conditions is met:

1. 180 seconds have elapsed since the beginning of the search.
2. 200 relaxations have been performed (with a maximum of 1,000 backtracks per relaxation).

Similarly to the other application chapter, the choice of LNS relaxation and the associated parameters was performed from experience. We have run experiments on a set of possible well known LNS relaxations while varying their parameters, the seed of the random generator, etc. The relaxation described earlier has obtained the best results on all instances. Hence, in order to ease the benchmarking process, we have selected this single relaxation strategy.

Table 8.1 shows the ratio of objectives (historical/optimized value) obtained. We can see that the cost of transitions between different paper types is on average significantly reduced. However, the variance over this objective ratio is high: the reduction of transition cost is really important on some instances but it decreases less on other instances (and not at all on three instances). The ratio of the energy cost however has a small variance. On most of the instances, LNS is able to reduce energy costs by around 22.5%. These results are promising but somewhat optimistic since it relies on a perfect knowledge of electricity future prices. Since forecasts can, by definition, be wrong, the gain could be reduced in practice.

| | Global | Energy | Transition |
|---|---|---|---|
| Average | 6.40 | 1.29 | 56.14 |
| Variance | 69.46 | 0.10 | 84,211.22 |

Table 8.1: Ratio of historical and optimized objective values (historical/optimized).

### 8.3.3 *Industrial Prototype and Reception*

We have designed a working prototype with a GUI for this problem. This prototype was well received by our industrial partners. Furthermore, as we have been able to test our prototype on real historical data, we have shown the potential gain in terms of electricity costs that our optimization procedure could bring to the paper production site. The gains we have been able to obtain were higher than those expected by our industrial partners. We have however stated that these gains did occur under several favorable conditions. First of all, we have had access to the exact prices of the electricity market while under real conditions, such framework should rely on forecast prices. Second, we have

not taken into account the interruptions that could occur in the case of unexpected events. Nevertheless, being able to obtain such reduction in terms of electricity costs was very satisfying for our partners. Furthermore, the large reduction of the transition costs between different paper types was largely appreciated since it was not expected at first.

## 8.4 FUTURE WORK

We have not discussed the problem of computing electricity price forecasts in this chapter. We have made the assumption that the price forecasts were reliable and therefore we have not proposed a stochastic model taking into account the variations that could occur on the price forecasts. Furthermore, we have not discussed in this chapter the possibility of re-planning an existing production plan according to a difference between the actual electricity prices and their forecasts.

Some other results of high interest would be to test the electricity price forecasts of the Enertop module of N-SIDE[1] to obtain a better estimate of the real energy gain. Indeed, this work used the exact electricity prices and the gains obtained would probably be lower by working with an imperfect estimation. It was unfortunately not possible to use the Enertop module in this work. It would have required to feed the forecast module with external features (weather forecast, etc.) that we do not have for the historical data.

For the models we have proposed, it would be interesting to use Variable Objective Large Neighborhood Search (VLNS) [Sch13b] in order to obtain a better pruning from our two terms composing the objective or to compute a Pareto front using a multi-objective LNS [SH13]. The CP model could also be extended with stocking costs computations [Hou+14] since it is not desirable to produce too early before the deadlines. It would also be of high interest to obtain real data on the paper pulp production part of the process. This would allow us to couple the paper machine model with the paper pulp model happening just before in the production process. This would allow an integrated optimization of the whole production.

Finally, another aspect that was not covered in this work is the possibility to vary the speed of the production line. The idea would be to decrease the speed of the production line – implying a reduction of the electricity costs – when electricity prices are high. The hygienic paper

1 http://energy.n-side.com/enertop-energy-flexibility-optimization/

manufacturing site in Belgium does not do such thing but could do it in practice. While the managers of the site were reticent at the idea of decreasing the speed of the production line, good results on historical data could convince them. For our model, this would require some serious changes or even a complete rewriting of them.

### CONCLUSION

In this chapter, we have described the problem of reducing energy costs in paper tissue production. To tackle this problem, we propose to reorganize a large part of the manufacturing process: the production of paper rolls from paper pulp. According to forecasts of electricity prices, paper rolls whose production require a larger amount of energy will be produced when prices are low. On the opposite, paper rolls whose production require a smaller amount of energy will be produced when prices are high.

The problem is subject to many constraints; an important one is the order book that translates into hard production deadlines. To represent the problem, we propose two CP models including all the constraints. The first model that we have proposed is a model for the paper machine, producing paper rolls starting from paper pulp. A second model represents the production of paper pulp obtained by mixing short and long wood fibers with water. These models are solved successively such that the optimized production plan obtained on the paper machine model will be an input to the paper pulp production model.

The deadline and stock constraints of the problem are expressed with Nested GCC that has been discussed in Chapter 4. The performances of our new propagation procedure was evaluated on instances generated from historical data. The results we obtained on these historical instances are similar to those that were obtained on randomly generated instances from Chapter 4. The conclusions about the new FWC propagation procedure, stating that it achieved larger pruning in smaller amounts of time, seem thus to be validated by those results.

The paper machine model was solved using an LNS approach and the objective values after optimization were compared to the historical ones. The gains on both objectives are substantial and this highlights the benefits of using our model to reduce electricity costs instead of deciding the production plan by hand. However, those results were obtained with full knowledge of the electricity prices instead of forecasts. There remains several opportunities to consider other aspects of

the problem that could potentially lead to a more robust model taking other parameters into account.

# CONCLUSION AND PERSPECTIVES

## CONCLUSION

In this thesis, we have been confronted to time-related problems. In these problems, the aim is to determine when a set of actions can be placed in time. As these problems are subject to many constraints, we have decided to use Constraint Programming approaches to solve them. Most of the time, we have decided to represent our problems as scheduling problems. The four main categories of scheduling abstractions allowed us to easily describe our problems in a concise and declarative way. We have also considered production planning problems and their associated models. Along this work, we have described two new propagation procedures. These propagation procedures were then used in several real-world applications.

### *Efficient Propagators*

During our modelization of various scheduling problems, we have discovered that there were no existing efficient propagation procedure to consider unary resource with transition times. We have thus developed a dedicated propagation procedure for this constraint. Our propagation procedure proposes to extend the existing unary resource propagators from Vilìm with small modifications on the $\Theta$-tree and $\Theta$-$\Lambda$-tree data structures. The propagation procedure we have proposed is efficient, offers significant additional pruning, and is faster than alternative existing propagators. It is also easy to implement in solvers already embedding the unary resource propagators from Vilìm.

We have also considered production planning problems. While being also time-related, they are not modeled as scheduling problems. Instead, we have modeled them with a set of unit periods (all of the same duration) to which we associate a type of production. While modeling a production planning problem, we have modeled stock and deadline constraints. To help the propagation procedure, we have proposed a FWC propagator for the Nested Global Cardinality Constraint. This propagator allows us to perform more pruning than a classic decomposition into multiple GCCs.

*Real-World Applications*

The two abstractions we have introduced were useful to tackle real-world problems that have been considered in this thesis. The two first problems considered patient scheduling in health care treatment centers. Several models were proposed for these problems and the promising results seem to indicate that there is a bright future for scheduling in the health care sector.

First, we have discussed the scheduling of patient treatment sessions in Proton Therapy centers. We have designed a CP model to tackle this problem. This model has been able to solve real-world sized instances with an accurate model. Second, we have considered the scheduling of patient appointments for their successive treatment sessions in a PT center. In this approach, we have dedicated our effort to obtain a *reactive* model. This approach allows a user to modify an existing appointment plan to add/remove constraints and obtain a new solution in very short amounts of time (less than 5 seconds). As this method has performed well, this proves that approaches combining CP and LNS can easily be used in a reactive context.

The second problem considered aimed at scheduling patient treatment sessions in a Nuclear Medicine center. This problem is complex since we have to deal with the management of radioactive resources. We have proposed a scheduling model for this problem that contained constraints for which there was no existing propagation procedures. However, using view-based propagator derivation techniques, we have been able to obtain propagation procedures using existing propagators with only a very small time overhead. This proves that even though some problems might seem complex, the expressibility of CP allows to model and solve them relatively easily.

Finally, the last application aims at reducing the energy costs of industrial processes. Electricity price forecasts determine the cost of unit consumption at each period. We have proposed a production planning model to shift the production of various products with different production energy demands. We have had access to historical data and we have been able to compare historical production plans to optimized ones. The versions obtained after optimization proposed non-negligible energy cost savings. Again, we have shown the practicability of a CP + LNS approach to solve large instances of this problem.

*Final Words*

All the problems we have considered in this thesis were optimization problems. Even though CP is useful in the case of optimization, its completeness can be a problem when considering large instances of complex problems. We have decided to couple CP with LNS to allow more diversification and to discover high quality solutions faster. This combination of CP with LNS has proven to work well in practice on the various problems tackled with real-world instances.

This whole thesis, through the abstractions introduced and the real-world problems that have been solved, has shown the expressive power of scheduling. More importantly, several propagation procedures for hard global constraints have been introduced. This adds even more tools to the Constraint Programming paradigm. In the case of scheduling, the combination of CP and LNS has proven to solve hard problems on large instances. This thesis is thus yet another proof of the efficiency of CP and LNS when tackling scheduling problems.

## PERSPECTIVES

This thesis has introduced two new propagation procedures and has tackled several hard real-world time-related problems. This work is the product of four years of work but many other perspectives remain to explore. We propose here to list some perspectives opened up by the content of this thesis.

### *Unary Resource with Transition Times*

We have presented a new propagation procedure for the unary resource with transition times by extending Vilìm's work. The proposed propagation procedure has however several weaknesses. The propagators introduced all rely on the computation of the ect of a set of activities. To include transition times, it adds the smallest sequence of transitions over *all the activities* whose length is equal to the cardinality of the considered set. It will thus fail to propose consequent pruning in the case of sparse transition times.

A recent work – not detailed in this thesis – has been submitted at the CP2016 conference to consider *families* of transitions. It proposes to group activities in families: subsets of activities with null transitions between each of them. The computation of the ect is no more based on the cardinality of a set of activities, but on the number of different families inside a set of activities.

This recent work does however not cover the case where transition times are non-null but of very different importance. In such cases, the very large transition times will not be included in the ect computation. A solution could be to consider families discovered by clustering means. This would allow to consider transitions between clusters of activities, permitting the inclusion of large transition times into the computation of the ect of a set of activities.

### *Reactive Optimization*

We have tackled the Ten Weeks Ahead Appointment Schedule Problem using *reactive optimization*. Reactive optimization aims at proposing solutions to a user in a very short amount of time while offering him the possibility to modify the model. Once a solution has been discovered, the user has the opportunity to both add and remove constraints from

the model. Once he is satisfied with its modified model, the solver has a very short amount of time to discover new feasible solutions to this model.

To achieve reactive optimization, we have proposed an approach combining CP and LNS. There exist many critical and industrial applications that could be solved using reactive optimization. The limits and extensions of our approach with CP and LNS could be studied and tested in several other real-world applications.

*Multi-Objective Optimization with CP*

We have proposed a multi-objective model for the Nuclear Medicine Problem. This model was solved with a CP + LNS approach. However, most CP models attempt to aggregate objectives nowadays. This is due to the complexity and search space expansion brought by the consideration of multiple objectives. However, we could use the specificities of the CP framework to largely reduce the search space of the problem. This could be done with the multi-objective LNS introduced in [SH13].

In most multi-objective applications, users are interested by only a small region of the solution space (i.e., a small portion of the Pareto front). This small region corresponds to acceptable tradeoffs between the objectives considered in the problem. Allowing the user to indicate the region of interest on a first approximation of the Pareto front – using reactive optimization – could largely reduce the search space. Indeed, bounding the objectives would allow propagators to remove large parts of the search space.

*Wrap-up*

Many other promising leads have been addressed in this thesis at the end of its chapters. With so many open questions and only so few time to investigate them, we can conclude that optimization still encloses an infinity of marvels that remains to be unearthed.

Part V

# APPENDIX

# A

# IMPLEMENTATION AND SOURCE CODE

*I'm sorry, Dave. I'm afraid I can't do that.*

—Hal 9000, *2001: A Space Odyssey*

*Unfortunately, no one can be told what the Matrix is. You'll have to see it for yourself.*

—Morpheus, *The Matrix*

*Welcome to the real world.*

—Morpheus, *The Matrix*

*One of my most productive days was throwing away 1000 lines of code.*

—Ken Thompson

*Talk is cheap. Show me the code.*

—Linus Torvalds

The reasearch described in this thesis has involved a huge implementation work. Behind every algorithm described in this thesis are hidden at least three different implementations into the open source solver OscaR [Osc12]. The implementation work was performed incrementally and strong efforts were applied to keep a low coupling in the code, make it clear, concise and easy to maintain. However, the various pieces of code we have implemented are – or are related to – propagators. Depending on the events to which they are registered, propagators can be called up to several times at every node of the search tree. Hence, their performances are critical and the source code implementing them should ensure computation times as low as possible.

Making a strongly uncoupled, clear and concise code implementation often comes in contradiction with low computation times. We have dedicated our efforts to find a tradeoff between these two contradictory objectives that seemed to put them in balance. The motto of the implementation we have performed along this thesis was thus the following one: "fast and clean" in opposition to the two other extremes: "fast and furious" and "slow and beautiful".

All the code of propagators that has been developed during this thesis is available in the OscaR open-source solver [Osc12]. If an attentive reader wishes to test our implementation, may he feel free to do so and report bugs or ideas of improvement should he find some.

We will not explain the steps to follow to install and use OscaR as these may have changed by the time you read these lines. To install and use the OscaR solver, please visit `http://oscarlib.bitbucket.org/` and `https://bitbucket.org/oscarlib/oscar/wiki/Home`. By the time you read these lines, the solver itself might have changed and therefore all the code of the models described in this work might not run on the last version of the solver. All the pieces of code that you will find in this thesis are based on the version 3.1.0 of the project. This version should be available online and it should therefore be possible to get it and run most pieces of code described here.

## A.1 CODE TESTING

Each of the code implementation we have developed was tested as much as possible with unit testing. As every people who has ever worked on critical pieces of code in a large ecosystem – such as propagators for a CP solver – knows, unit testing alone does not guaran-

tee that the code does not contain any bug. We have therefore implemented the following testing procedure to test the correct implementation of our propagators.

All the propagators we have developed in this work are global propagators that can be expressed with decompositions of small binary constraints. We thus proceeded to write two models implementing the same constraints. In the first model $M_b$, the considered constraint is expressed with the decomposition of small binary constraints. The second model $M_g$ is the model $M_b$ to which our global propagator has been added. Both models were run on small instances randomly generated. At the end of each run, we have tested that the following conditions where respected during the whole search:

SMALLER OR EQUIVALENT NUMBER OF NODES
As the pruning achieved in $M_g$ should be larger than – or at least equivalent to – the one in $M_b$, the number of nodes should be smaller – or equivalent – for $M_g$.

SMALLER OR EQUIVALENT NUMBER OF BACKTRACKS
Similarly to the number of nodes, the number of backtracks for $M_g$ should be smaller or equivalent than for $M_b$.

EQUAL NUMBER OF SOLUTIONS
While the binary decomposition model $M_b$ should achieve a smaller pruning than $M_g$, it has the same deductive power when it comes to checking a complete assignment. Hence, the number of solutions should be equivalent for both models. If this is not the case, this means that $M_g$ has discarded feasible solutions and therefore is not correct.

This random instance generated testing strategy is correct if two conditions are met:

THE BINARY DECOMPOSITION IS CORRECT
If the binary decomposition is not correct, this testing does not make any sense. Two sources of mistakes can happen. First, the propagation of the binary constraint used could be incorrect. As these were already heavily tested and validated on a large number of tests, instances and application in the OscaR solver, we have assumed that this was not the case. Second, the decomposition of the global constraint into binary constraint has been wrongly encoded. This has happened once or twice, but it is easy to find mistakes in such case

when comparing solutions on very small instances that can be solved by hand.

THE RANDOM INSTANCES ARE REPRESENTATIVE

This second condition is usually incorrect. While there has been a lot of attempts in coding history to achieve representative testing[1], it was not applicable in the case of global propagators. Hence, it has happened several times that even though our testing procedure validated our implementation, we discovered later an instance in which it produced erroneous results. Indeed, as a lot of data structures, such as reversibles, sparse sets, etc., are implied in the implementation of global propagators, it can fail in very specific cases that are not easily represented in small instances. Every time we have encountered a bug in our implementation that was not detected by our tests, we extracted the source of error and put it in a test that was added to the original test suite.

## A.2 A SIMPLE MODEL WITH OSCAR CP

Writing a CP model in the OscaR open source solver [Osc12] is easy whether or not you have some knowledges in Scala. One of the main advantages of the Scala programming language, is that it leaves many liberties to developers to write a Domain-Specific Language (DSL). The OscaR CP module has developed its own light DSL allowing models to be both easy to write and easy to read. We propose here a small example of model written in the OscaR CP module. This example illustrates different pieces of code that will not be shown in further models such as the import section, how to declare variables, etc. The example is a classic N-Queens CP model.

The N-Queens problem is as follows: considering a square chess board of $n \times n$ tiles, one has to place $n$ queens on the board such that they do not attack each other. A queen attacks other pieces on the same line, on the same column and on the same diagonals (upwards and downwards). A classic modelization of this problem associates one variable $q_i$ to each of the queens to be placed on the board. As queens cannot be placed in a same column, each variable $q_i$ represents the position of a queen on the column $i$. The value associated to one variable thus becomes the row at which the corresponding queen is placed. Then, to impose that no two queens can be on the same

1 A good example is the ScalaCheck framework for the Scala language [Nil14].

row, an `AllDifferent` constraint is imposed on all the variables. To impose that no two queens can be on the same upward diagonal, an `AllDifferent` constraint is imposed on view variables $q_i + i$ for all $i \in [0, n-1]$. The same reasoning is applied for downward diagonal and an `AllDifferent` constraint is imposed on view variables $q_i - i$ for all $i \in [0, n-1]$.

An OscaR model for the N-Queens problem is shown in Code 1. At the top of the code, we import the CP module of the OscaR solver and all the CP abstractions needed for the model. As the `NQueens` object (i.e., singleton class in Scala) extends `CPModel`, it is defined as a subclass of `CPModel`. This has the advantage that it defines a CP solver as an implicit value[2]; all the methods that can be applied on – or that need as a parameter – this implicit can be declared without referring to the CP solver in itself. The code defines the number of queens $n$ considered in the model; here $n = 10$. The variables of the model are instantiated as the value `queens`. They are instantiated as an array of $n$ variables (indexed from 0 to $n-1$) and their domains are initialized to $[0, n-1]$. The three `allDifferent` constraints mentioned earlier are then posted with the `add` function. The search is defined as a binary first fail strategy on all the variables `queens`. Finally, the `start()` instruction begins the resolution of the model. When no parameters are provided, the `start()` instruction performs a complete exploration of the search space i.e., all feasible solutions are searched.

As one can see in this example, the definition of a model, a search strategy and the run of its resolution is simple, clear and concise in the OscaR CP module. With the help of only ten lines of code, we were able to full define declaratively a simple model. The source code of the model itself is very easy to read and understand – it is *almost* a plain English declaration of the model.

2 For more information about implicits in Scala, refer to [OSV08].

```
import oscar.cp._

object NQueens extends CPModel with App {
  val nQueens = 10 // Number of queens
  val Queens = 0 until nQueens

  // Variables
  val queens = Array.fill(nQueens)(CPIntVar.sparse(0, nQueens - 1))

  // Constraints
  add(allDifferent(queens))
  add(allDifferent(Queens.map(i => queens(i) + i)))
  add(allDifferent(Queens.map(i => queens(i) - i)))

  // Search heuristic
  search(binaryFirstFail(queens))

  // Execution
  val stats = start()
  println(stats)
}
```

Code 1: Model for the N-Queens problem implemented in the OscaR solver.

## A.3 NESTED GCC

In Chapter 4, we propose a method to strengthen the bounds passed as arguments to the Nested GCC. This is done in three main steps. These three steps are performed sequentially at the setup of the constraint:

### PER-VALUE DEDUCTIONS

*Per-value deduction* considers a single value with respect to all the variables. It also fills the bounds for the ranges that were not described in the initial bounds. Code 2 is a simplified version of the per-value deduction code from OscaR. The main loop goes through all the values `vi` and the forward and backward updates are performed inside of it. The first internal `while` loop iterates over increasing `i` indices and performs the forward update for both lower and upper bounds. Similarly, the second internal loop does the backward update over decreasing `i` indices.

```
1  def perValueDeductions() {
2    var vi = 0
3    while (vi < nValues) {
4      var i = 0
5      // Forward update
6      while (i < nVariables) {
7        i += 1
8        lower(vi)(i) = math.max(lower(vi)(i), lower(vi)(i - 1))
9        upper(vi)(i) = math.min(upper(vi)(i), upper(vi)(i - 1) + 1)
10     }
11     // Backward update
12     while (i > 0) {
13       i -= 1
14       lower(vi)(i) = math.max(lower(vi)(i), lower(vi)(i + 1) - 1)
15       upper(vi)(i) = math.min(upper(vi)(i), upper(vi)(i + 1))
16     }
17     vi += 1
18   }
19 }
```

Code 2: Per-value deductions to strengthen the bounds of the Nested GCC.

### INTER-VALUE DEDUCTIONS

*Inter-value deduction* considers a single range of variables with respect

to all the values bounded over this range. Code 3 is a simplified version of the inter-value code from OscaR. The main loop goes through all the variable ranges `i`. Inside of it, a first loop computes the sum of all the lower bounds and the sum of all the upper bounds. Then, a second loop updates the bounds using the previously computed sums for inter-value deductions. Note that this function return type is `CPOutcome`. Indeed, while browsing the bounds, this function detects if they are consistent. There are two reasons for which this could fail: either the upper bound of a value is lower than its lower bound for a given range, or the sum of the lower bounds over a range exceeds the number of variables within the range, and the opposite for the sum of upper bounds.

#### REDUCTION TO A MINIMAL SET OF BOUNDS

Once the two bound strengthening steps have been performed, the bounds can be reduced to a minimal set of useful bounds. Code 4 is a simplified version of the code from OscaR. It first defines two filter functions. The first one, `flatFilter`, removes all the bounds that are in the middle of a *plateau*. The second filter, `slopeFilter`, removes the bounds that are in the middle of an increasing/decreasing slope. At the end of the function, a `while` loop applies these filters for both lower and upper bounds for all the possible values `vi`.

```
def interValueDeductions(): CPOutcome = {
  var i = 1
  while (i < nVariables) {
    // Compute the sums of the all values on range 0 to i
    var lowerSum, upperSum, vi = 0
    while (vi < nValues) {
      // The lower bound cannot be higher than the upper bound
      if (lower(vi)(i) > upper(vi)(i)) {
        return Failure
      }
      lowerSum += lower(vi)(i)
      upperSum += upper(vi)(i)
      vi += 1
    }
    // Test the sums
    if (lowerSum > i || upperSum < i) {
      return Failure
    }
    // Update of the bounds
    vi = 0
    while (vi < nValues) {
      lower(vi)(i) = math.max(lower(vi)(i), i - upperSum + upper(vi)(i))
      upper(vi)(i) = math.min(upper(vi)(i), i - lowerSum + lower(vi)(i))
      vi += 1
    }
    i += 1
  }
  Suspend
}
```

Code 3: Inter-value deductions to strengthen the bounds of the Nested GCC.

```
1  private def minimalSetOfBounds() {
2    // Bound filtered if equal to previous and next bound
3    def flatFilter(prevIdx: Int, prevVal: Int,
4                   nextIdx: Int, nextVal: Int) = {
5      nextVal > prevVal
6    }
7    // Bound filtered if in the middle of an in/decreasing slope
8    def slopeFilter(prevIdx: Int, prevVal: Int,
9                    nextIdx: Int, nextVal: Int) = {
10     nextVal - prevVal < nextIdx - prevIdx
11   }
12
13   var vi = 0
14   while (vi < nValues) {
15     filterGeneric(lower(vi), filterSlope, filterFlat)
16     filterGeneric(upper(vi), filterFlat, filterSlope)
17     vi += 1
18   }
19 }
```

Code 4: Reduction to a minimal set of bounds after bounds strengthening of the Nested GCC.

## A.4 UNARY RESOURCE WITH TRANSITION TIMES

The work from Chapter 5 has lead to many prototypes and implementation variations. A clean version of the constraint is provided in OscaR. We propose here to provide a simplified version of the code of the many components implemented in OscaR.

### A.4.1 *Transitions in a Set of Activities*

In Section 5.3, we have described several relaxations to compute a lower bound of the transitions occurring in a set of activities based only on its cardinality. We propose here to provide a subset of their implementation.

In Code 5, we present the dynamic programming relaxation implementation similar to the one from OscaR. The double indexed array `D(m)(u)` represents the shortest walk of length exactly `m` from the source to `u`. The three nested `for` loops compute this value for all the `m` and all the `u`. To obtain $\underline{\mathrm{tt}}(m)$, we browse for all `m` the walks over `u` and we keep the minimal value encountered.

Another approach presented in Section 5.3 proposes to use a minimum cost flow relaxation. A simple implementation of this relaxation is shown in Code 6. In this implementation, we use the `oscar-linprog` package that has a wrapper around the solver *LPSolve*. After initialization, the main `for` loop solves `n` models, one for each $k$ to get the value of $\underline{\mathrm{tt}}(k)$.

```scala
def dynamicProgramming(ttMatrix: Array[Array[Int]]): Array[Int] = {
  val n = ttMatrix.length
  // D(m)(u) is the shortest walk of length exactly m from source to u.
  val D = Array.fill(n, n)(Int.MaxValue)
  // Setting initial values
  for (u <- 0 until n) {
    D(0)(u) = 0
  }
  // Dynamic Programming
  for (m <- 0 until n - 1) {
    for (u <- 0 until n) {
      for (v <- 0 until n) {
        if (u != v) {
          val alt = D(m)(v) + ttMatrix(v)(u)
          if (D(m + 1)(u) > alt) {
            D(m + 1)(u) = alt
          }
        }
      }
    }
  }
  val aggregatedResults = Array.fill(n)(Int.MaxValue)
  for (m <- 0 until n) {
    for (u <- 0 until n) {
      aggregatedResults(m) = math.min(aggregatedResults(m), D(m)(u))
    }
  }
  aggregatedResults
}
```

Code 5: Relaxation of transitions in a set of activities per cardinality using a dynamic programming approach.

```scala
import oscar.algebra._
import oscar.linprog.modeling._
import oscar.linprog.interface.lpsolve.LPSolveLib

def minAssignmentFlow(ttMatrix: Array[Array[Int]]): Array[Int] = {
  val n = ttMatrix.length
  val bestPossibleBounds = Array.fill(n)(0)
  val modifiedTT = Array.tabulate(n, n)((i, j) => {
    if(i != j) ttMatrix(i)(j)
    else Int.MaxValue
  })

  for (k <- 1 until n) {
    implicit val lpModel = new MPModel(LPSolveLib)
    implicit val solver = lpModel.solver
    val x = Array.tabulate(n, n)((i, j) =>
      MPFloatVar("x" + (i, j), 0.0, 1.0))

    minimize(oscar.algebra.sum(0 until n, 0 until n)(
      (i, j) => modifiedTT(i)(j) * x(i)(j)
    ))

    add(oscar.algebra.sum(0 until n, 0 until n)((i, j) =>
      x(i)(j)} =:= k)

    for (i <- 0 until n) {
      add(oscar.algebra.sum(0 until n){j => x(i)(j)} <:= 1)
      add(oscar.algebra.sum(0 until n){j => x(j)(i)} <:= 1)
    }

    solver.solve
    bestPossibleBounds(k) = solver.objectiveValue.get.toInt
    solver.release
  }

  bestPossibleBounds
}
```

Code 6: Relaxation of transitions in a set of activities per cardinality using a minimum cost flow approach.

### A.4.2 *Propagators*

In Section 5.5, we have described several propagators that can be used for the unary resource with transition times. We propose here to provide a subset of their implementation. All these propagators rely on the `ThetaLambdaTreeTT` data structure.

An implementation of the Detectable Precedences algorithm from Section 5.5 is shown in Code 7. This implementation is simplified to ease the reading of the code. It begins by clearing the Θ-tree `tree` whose state might have been altered by other algorithms. Then it sorts two arrays: `orderedMaxStartIds` and `orderedMinEndIds`. These arrays contain indices that will be passed by reference to obtain sorted elements. The main `while` loop iterates over the activities ordered by non-decreasing ect. The internal `while` loop adds activities in `tree` while precedences are detected. Then, the ect of the tree is retrieved and added to the minimal transition to current activity to obtain a new value for its min start: `newStartMins(ectIndex)`. Finally, the domains of the start variables are updated by the `updateECTs(newStartMins)` instruction.

The Edge Finding algorithm from Section 5.5 uses a Θ-Λ-tree data structure. Our implementation of `ThetaLambdaTreeTT` can be used for both Θ-tree and Θ-Λ-tree structures. A simplified version of the Edge Finding implementation from OscaR is shown in Code 8. It begins by filling `tree` with all the activities and then it sorts activities by non-decreasing lct. Then its main `while` loop goes through the activities by non-increasing lct. Inside the loop, if the Overload Checking condition is verified, a failure is returned. Then, the current activity is grayed. The internal `while` loop is repeated as long as the $\overline{\text{ect}}$ of the tree is larger than the lct of the current activity. In this loop, the gray activity responsible for $\overline{\text{ect}}$ is retrieved; its index is `estIndex`. Then a new value for the min start of the responsible activity is computed: `newStartMins(estIndex)`. It is then removed from the `tree`. At the end of the function, the domains of the start variables are updated by the `updateECTs(newStartMins)` instruction.

```
def detectablePrecedences(startMins: Array[Int], startMaxs: Array[Int],
                          endMins: Array[Int], newStartMins: Array[Int],
                          tree: ThetaLambdaTreeTT,
                          minTTToActi: Array[Int]): CPOutcome = {
  // Clearing the tree
  mergeSort(orderedMinStartIds, startMins)
  tree.clearAndPlaceLeaves(startMins, minDurations)

  // Sorting activities in non-decreasing lst
  mergeSort(orderedMaxStartIds, startMaxs)

  // Sorting activities in non-decreasing ect
  mergeSort(orderedMinEndIds, endMins)

  var i, q = 0
  while (i < nTasks) {
    val ectIndex = orderedMinEndIds(i)
    while (q < nTasks &&
           endMins(ectIndex) > startMaxs(orderedMaxStartIds(q))) {
      tree.insert(orderedMaxStartIds(q))
      q += 1
    }
    newStartMins(ectIndex) =
      math.max(newStartMins(ectIndex),
               tree.ectWithoutActi(ectIndex) + minTTToActi(ectIndex))
    i += 1
  }

  // Updating the domains of start variables
  updateECTs(newStartMins)
}
```

Code 7: Detectable Precedences propagation algorithm for the unary resource with transition time constraint.

```
def edgeFinding(startMins: Array[Int], endMaxs: Array[Int],
                endMins: Array[Int], newStartMins: Array[Int],
                tree: ThetaLambdaTreeTT,
                minTTToActi: Array[Int]) : CPOutcome = {
  // Inserting all activities in the tree
  mergeSort(orderedMinStartIds, startMins)
  tree.fillAndPlaceLeaves(orderedMinStartIds, startMins, minDurations)

  // Sorting activities in non-decreasing lct
  // This array will be browsed from right to left
  mergeSort(orderedMaxEndIds, endMaxs)

  var estIndex = 0
  var j = nTasks - 1
  while (j > 0) {
    if(tree.ect._1 > endMaxs(orderedMaxEndIds(j))) {
      return CPOutcome.Failure
    }

    tree.grayActivity(orderedMaxEndIds(j))

    j -= 1
    while(tree.ectBar > endMaxs(orderedMaxEndIds(j))) {
      estIndex = orderedMinStartIds(tree.responsibleECTBar)
      newStartMins(estIndex) =
        math.max(newStartMins(estIndex),
                 tree.ect + minTTToActi(estIndex))
      tree.remove(estIndex)
    }
  }

  updateECTs(newStartMins)
}
```

Code 8: Edge Finding propagation algorithm for the unary resource with transition time constraint.

# BIBLIOGRAPHY


[ALN99] Fouad Ben Abdelaziz, Pascal Lang, and Raymond Nadeau. "Dominance and Efficiency in Multicriteria Decision under Uncertainty." English. In: *Theory and Decision* 47 (3 1999), pp. 191–211.

[AB93] Abderrahmane Aggoun and Nicolas Beldiceanu. "Extending CHIP in Order to Solve Complex Scheduling and Placement Problems." In: *Mathematical and computer modelling* 17.7 (1993), pp. 57–73.

[All+08] Ali Allahverdi, Chi to Daniel Ng, Tai Chiu Edwin Cheng, and Mikhail Kovalyov. "A Survey of Scheduling Problems with Setup Times or Costs." In: *European Journal of Operational Research* 187.3 (2008), pp. 985–1032.

[Anj56] Shahroz Anjum. "Multi Objective Economic Emission Dispatch Using Modified Multi Objective Particle Swarm Optimization." PhD thesis. THAPAR UNIVERSITY, 1956.

[Apt03] Krzysztof Apt. *Principles of Constraint Programming*. UK: Cambridge University Press, 2003.

[ABF04] Christian Artigues, Sana Belmokhtar, and Dominique Feillet. "A New Exact Solution Algorithm for the Job Shop Problem with Sequence-Dependent Setup Times." In: *Integration of ai and or techniques in constraint programming for combinatorial optimization problems*. Springer, 2004, pp. 37–49.

[ABF05] Christian Artigues, Fabrice Buscaylet, and Dominique Feillet. "Lower and Upper Bound for the Job Shop Scheduling Problem with Sequence-Dependent Setup Times." In: *Proceedings of the second multidisciplinary international conference on scheduling: theory and applications (MISTA'2005)*. 2005.

[AF08] Christian Artigues and Dominique Feillet. "A Branch and Bound Method for the Job-Shop Problem with Sequence-Dependent Setup Times." In: *Annals of Operations Research* 159.1 (2008), pp. 135–159.

[BSV08] Egon Balas, Neil Simonetti, and Alkis Vazacopoulos. "Job Shop Scheduling with Setup Times, Deadlines and Precedence Constraints." In: *Journal of Scheduling* 11.4 (2008), pp. 253–262.

[Bap96] Philippe Baptiste. "Disjunctive Constraints for Manufacturing Scheduling: Principles and Extensions." In: *International Journal of Computer Integrated Manufacturing* 9 (1996), pp. 306–310.

[BL96] 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*. Vol. 335. 1996, pp. 339–345.

[BLN01] Philippe Baptiste, Claude Le Pape, and Wim Nuijten. *Constraint-Based Scheduling: Applying Constraint Programming to Scheduling Problems*. New York, USA: Springer Science & Business Media, 2001.

[BC02] Nicolas Beldiceanu and Mats Carlsson. "A New Multi-Resource Cumulatives Constraint with Negative Heights." In: *International Conference on Principles and Practice of Constraint Programming*. Springer Berlin Heidelberg. 2002.

[Bel+96] Nicolas Beldiceanu, E. Bourreau, D. Rivreau, and Helmut Simonis. "Solving Resource-Constrained Project Scheduling Problems with CHIP." In: *Fifth International Workshop on Project Management and Scheduling, Poznan, Poland* (1996).

[Bel+15] Nicolas Beldiceanu, Mats Carlsson, Pierre Flener, Xavier Lorca, Justin Pearson, Thierry Petit, and Charles Prud'Homme. "A Modelling Pearl with Sortedness Constraints." In: *Proceedings of the Global Conference on Artificial Intelligence*. 2015.

[Bel58] Richard Bellman. "On a Routing Problem." In: *Quarterly of Applied Mathematics* 16.1 (1958).

[Bru99] Peter Brucker. "Complex Scheduling Problems." In: *Zeitschrift Operations Research* 30 (1999), p. 99.

[BT96] Peter Brucker and Olaf Thiele. "A Branch & Bound Method for the General-Shop Problem with Sequence Dependent Setup-Times." In: *Operations-Research-Spektrum* 18.3 (1996), pp. 145–161.

[CP89] Jacques Carlier and Éric Pinson. "An Algorithm for Solving the Job-Shop Problem." In: *Management Science* 35.2 (1989), pp. 164–176.

[CLS15] Sascha Van Cauwelaert, Michele Lombardi, and Pierre Schaus. "Understanding the Potential of Propagators." In: *Integration of AI and OR Techniques in Constraint Programming*. Springer, 2015, pp. 427–436.

[Cau+16] Sascha Van Cauwelaert, Cyrille Dejemeppe, Jean-Noël Monette, and Pierre Schaus. "Efficient Filtering for the Unary Resource with Family-based Transition Times." In: *Principles and Practice of Constraint Programming (submitted, to be accepted)*. Springer International Publishing, 2016.

[CSP12] Simon Cherry, James Sorenson, and Michael Phelps. *Physics in Nuclear Medicine*. Makati City, Philippines: Elsevier Health Sciences, 2012.

[Cle+10] Alexis De Clercq, Thierry Petit, Nicolas Beldiceanu, and Narendra Jussien. "A Soft Constraint for Cumulative Problems with Over-Loads of Resource." In: *Doctoral Program of the 16th International Conference on Principles and Practice of Constraint Programming (CP'10)*. 2010, pp. 49–54.

[Coo+11] William Cook, William Cunningham, William Pulleyblank, and Alexander Schrijver. *Combinatorial Optimization*. Wiley Series in Discrete Mathematics and Optimization. Toronto, Canada: Wiley, 2011.

[Dej12] Cyrille Dejemeppe. "Cumulative Job-Shop Scheduling for Proton Therapy." Louvain-La-Neuve, Belgium: ICTeam, Université Catholique de Louvain, 2012.

[Dej13] Cyrille Dejemeppe. "Alternative Job-Shop Scheduling For Proton Therapy." In: *CP Doctoral Program 2013* (2013), p. 67.

[DCS15] Cyrille Dejemeppe, Sascha Van Cauwelaert, and Pierre Schaus. "The Unary Resource with Transition Times." In: *Principles and Practice of Constraint Programming*. Cork, Ireland: Springer International Publishing, 2015.

[DD14] Cyrille Dejemeppe and Yves Deville. "Continuously Degrading Resource and Interval Dependent Activity Durations in Nuclear Medicine Patient Scheduling." In: *Integration of AI and OR Techniques in Constraint Programming*. Cork, Ireland: Springer International Publishing, 2014, pp. 284–292.

[DSD15] Cyrille Dejemeppe, Pierre Schaus, and Yves Deville. "Derivative-Free Optimization: Lifting Single-Objective to Multi-Objective Algorithm." In: *Integration of AI and OR Techniques in Constraint Programming*. Barcelona, Spain: Springer International
Publishing, 2015, pp. 124–140.

[Dej+16] Cyrille Dejemeppe, Olivier Devolder, Victor Lecomte, and Pierre Schaus. "Forward-Checking Filtering for Nested Cardinality Constraints: Application to an Energy Cost-Aware Production Planning Problem for Tissue Manufacturing." In: *Integration of AI and OR Techniques in Constraint Programming*. Banff, Canada: Springer International Publishing, 2016.

[DM02] Elizabeth Dolan and Jorge Moré. "Benchmarking Optimization Software with Performance Profiles." In: *Mathematical Programming* 91.2 (2002), pp. 201–213.

[DK13] Christoph Dürr and Sigrist Knust. *The Scheduling Zoo: A Searchable Bibliography in Scheduling*. 2013.

[Elm68] Salah Elmaghraby. "The Machine Sequencing Problem – Review and Extensions." In: *Naval Research Logistics Quarterly* 15.2 (1968), pp. 205–232.

[FLN00] Filippo Focacci, Philippe Laborie, and Wim Nuijten. "Solving Scheduling Problems with Setup Times and Alternative Resources." In: *AIPS*. 2000, pp. 92–101.

[FLM02] Filippo Focacci, Andrea Lodi, and Michela Milano. "A Hybrid Exact Algorithm for the TSPTW." In: *INFORMS Journal on Computing* 14.4 (2002), pp. 403–417.

[FI02] Joanna Fowler and Tatsuo Ido. "Initial and Subsequent Approach for the Synthesis of 18FDG." In: *Seminars in Nuclear Medicine* 32.1 (2002). Impact of FDG-PET Imaging on the Practice of Medicine, pp. 6 –12.

[GPG01] Caroline Gagné, Wilson Price, and Marcel Gravel. *Scheduling a Single Machine with Sequence Dependent Setup Time Using Ant Colony Optimization*. Montréal, Canada: Faculté des sciences de l'admi-nistration de l'Université Laval, Direction de la recherche, 2001.

[GHPS15] Steven Gay, Renaud Hartert, and Christophe Lecoutre and Pierre Schaus. "Conflict Ordering Search for Scheduling Problems." In: *Principles and Practice of Constraint Programming*. Springer, 2015, pp. 140–148.

[Glo86] Fred Glover. "Future Paths for Integer Programming and Links to Artificial Intelligence." In: *Computers & Operations Research* 13.5 (1986), pp. 533–549.

[GVV08] Miguel González, Camino Vela, and Ramiro Varela. "A New Hybrid Genetic Algorithm for the Job Shop Scheduling Problem with Setup Times." In: *ICAPS*. 2008, pp. 116–123.

[Gra+79] Ronald Graham, Eugene Lawler, Jan Karel Lenstra, and AHG Rinnooy Kan. "Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey." In: *Annals of Discrete Mathematics* 5 (1979), pp. 287–326.

[GH10] Diarmuid Grimes and Emmanuel Hebrard. "Job Shop Scheduling with Setup Times and Maximal Time-Lags: A Simple Constraint Programming Approach." In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer, 2010, pp. 147–161.

[Gro12] PTCOG (Particle Therapy Co-Operative Group). *Particle Therapy Facilities In Operation*. http://ptcog.web.psi.ch/ptcentres.html. 2012.

[Hou+14] Vinasétan Ratheil Houndji, Pierre Schaus, Laurence Wolsey, and Yves Deville. "The StockingCost Constraint." In: *Principles and Practice of Constraint Programming*. Springer. 2014, pp. 382–397.

[IBA12] IBA. *Proton Therapy*. http://www.iba-protontherapy.com/. 2012.

[Kel60] James Kelley Jr. "The Cutting-Plane Method for Solving Convex Programs." In: *Journal of the Society for Industrial and Applied Mathematics* (1960), pp. 703–712.

[Kir83] Scott Kirkpatrick. "Optimization by Simulated Annealing." In: *Science* 220.4598 (1983), pp. 671–679.

[KS97] Rainer Kolisch and Arno Sprecher. "Psplib - A Project Scheduling Problem Library." In: *European Journal of Operational Research*. 1997. Chap. 96, pp. 205–216.

[KRK08] Joshua Kollat, Patrick Reed, and Joseph Kasprzyk. "A New Epsilon-Dominance Hierarchical Bayesian Optimization Algorithm for Large Multiobjective Monitoring Network Design Problems." In: *Advances in Water Resources* 31.5 (2008), pp. 828–845.

[KDVk00] Panos Kouvelis, Richard Daniels, and George Vairaktarakis. "Robust Scheduling of a Two-Machine Flow Shop with Uncertain Processing Times." In: *IIE Transactions* 32.5 (2000), pp. 421–432.

[Kru56] Joseph Kruskal. "On the Shortest Spanning Subtree of a Graph and the Traveling Salesman Problem." In: *Proceedings of the American Mathematical society* 7.1 (1956), pp. 48–50.

[Lab09] Philippe Laborie. "IBM ILOG CP Optimizer for Detailed Scheduling Illustrated on Three Problems." In: *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer. 2009, pp. 148–162.

[LG07] Philippe Laborie and Daniel Godard. "Self-Adapting Large Neighborhood Search: Application to Single-Mode Scheduling Problems." In: *Proceedings MISTA-07* (2007), pp. 276–284.

[LS07] Mikael Lagerkvist and Christian Schulte. "Advisors for Incremental Propagation." In: *Principles and Practice of Constraint Programming–CP 2007*. Springer, 2007, pp. 409–422.

[Lau78] Jena-Lonis Lauriere. "A Language and a Program for Stating and Solving Combinatorial Problems." In: *Artificial Intelligence* 10.1 (1978), pp. 29–127.

[LW66] Eugene Lawler and David Wood. "Branch-and-Bound Methods: A Survey." In: *Operations Research* 14.4 (1966), pp. 699–719.

[Lev+05] W. Levin, H. Kooy, J. Loeffler, and T. DeLaney. "Proton Beam Therapy." In: *British Journal Of Cancer*. 2005, 849–854.

[Lit+63] John Little, Katta Murty, Dura Sweeney, and Caroline Karel. "An Algorithm for the Traveling Salesman Problem." In: *Operations Research* 11.6 (1963), pp. 972–989.

[MDH11] Jean-Baptiste Mairy, Yves Deville, and Pascal Van Hentenryck. "Reinforced Adaptive Large Neighborhood Search." In: *The Seventeenth International Conference on Principles and Practice of Constraint Programming (CP 2011)*. Perugia, Italy, 2011, p. 55.

[Mon10] Jean-Noël Monette. "Solving Scheduling Problems from High-Level Models." PhD thesis. Louvain-La-Neuve: Université Catholique de Louvain, Dept. INGI, 2010.

[Moo59] Edward Moore. "The Shortest Path Through a Maze." In: *Proceedings International Symposium Switching Theory*. 1959.

[Myc55] Jan Mycielski. "Sur le Coloriage des Graphes." In: *Colloquium Mathematicae*. Vol. 2. 3. 1955, pp. 161–162.

[Nil14] Rickard Nilsson. *ScalaCheck*. California, USA, 2014.

[OR-10] OR-Tools Team, Laurent Perron. *OR-TOOLS*. Available at `https://developers.google.com/optimization/`. 2010.

[ORS10] Angelo Oddi, Riccardo Rasconi, and Amedeo Cesta and Stephen Smith. "Exploiting Iterative Flattening Search to Solve Job Shop Scheduling Problems with Setup Times." In: *PlanSIG2010* (2010), p. 133.

[OSV08] Martin Odersky, Lex Spoon, and Bill Venners. *Programming in Scala*. California, USA: Artima, 2008.

[Osc12] Oscar Team, Pierre Schaus. *OscaR: Scala in OR*. Available from `bitbucket.org/oscarlib/oscar`. 2012.

[PS82] Christos Papadimitriou and Kenneth Steiglitz. *Combinatorial Optimization: Algorithms and Complexity*. Dover Books on Computer Science Series. Mineola, USA: Dover Publications, 1982.

[Par74] Vilfredo Pareto. *De l'économie mathématique*. Genève, Suisse: Librairie Droz, 1974.

[PP09] Thierry Petit and Emmanuel Poder. "The soft cumulative constraint." In: *arXiv preprint arXiv:0907.0939* (2009).

[Pie15] Pierre Schaus. *CP for the Impatient*. Available from `http://info.ucl.ac.be/ pschaus`. 2015.

[Pin12] Michael Pinedo. *Scheduling - Theory, Algorithms, and Systems*. Fourth Edition. New York, USA: Springer, 2012.

[QW06] Claude-Guy Quimper and Toby Walsh. "Global Grammar Constraints." English. In: *Principles and Practice of Constraint Programming - CP 2006*. Vol. 4204. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2006, pp. 751–755.

[RC15] Prakash Rao and Chaitanya. "Resource Constrained Project Scheduling Problems-A Review Article." In: *International Journal of Science and Research* 4.3 (2015), pp. 1509–1512.

[Rég96] Jean-Charles Régin. "Generalized Arc Consistency for Global Cardinality Constraint." In: *Proceedings of the Thirteenth National Conference on Artificial Intelligence - Volume 1*. AAAI'96. Portland, Oregon: AAAI Press, 1996, pp. 209–215.

[RVW06] Francesca Rossi, Peter Van Beek, and Toby Walsh. *Handbook of Constraint Programming*. Amsterdam, Netherlands: Elsevier, 2006.

[RN95] Stuart Russell and Peter Norvig. *Artificial Intelligence: A Modern Approach*. Upper Saddle River, USA: Prentice Hall, 1995.

[SM+13] Vianney le Clément de Saint-Marcq, Pierre Schaus, Christine Solnon, and Christophe Lecoutre. "Sparse-Sets for Domain Implementation." In: *Techniques for implementing constraint programming systems (TRICS) workshop at CP 2013*. Uppsala, Sweden, 2013.

[Sch13a] Pierre Schaus. "Variable Objective Large Neighborhood Search: A Practical Approach to Solve Over-Constrained Problems." In: *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*. 2013, pp. 971–978.

[Sch13b] Pierre Schaus. "Variable Objective Large Neighborhood Search: A Practical Approach to Solve Over-Constrained Problems." In: *Tools with Artificial Intelligence (ICTAI), 2013 IEEE 25th International Conference on*. IEEE. Washington D.C., USA, 2013, pp. 971–978.

[SH13] Pierre Schaus and Renaud Hartert. "Multi-Objective Large Neighborhood Search." In: *Principles and Practice of Constraint Programming*. Springer. Uppsala, Sweden, 2013, pp. 611–627.

[SS08] Christian Schulte and Peter Stuckey. "Efficient Constraint Propagation Engines." In: *Transactions on Programming Languages and Systems* 31.1 (Dec. 2008), 2:1–2:43.

[Sch+03] Heiko Schöder, Yusuf Erdi, Steven Larson, and Henry Yeung. "PET/CT: a new imaging technology in nuclear medicine." English. In: *European Journal of Nuclear Medicine and Molecular Imaging* 30.10 (2003), pp. 1419–1437.

[SW11] Robert Sedgewick and Kevin Wayne. *Algorithms*. Fourth Edition. Boston, USA: Addison-Wesley, 2011.

[Sha98] Paul Shaw. "Using Constraint Programming and Local Search Methods to Solve Vehicle Routing Problems." In: *Principles and Practice of Constraint Programming*. Pisa, Italy: Springer, 1998, pp. 417–431.

[Shi+79] William Shipley, Joel Tepper, George Prout, Lynn Verhey, Oscar Mendiondo, Michael Goitein, Andreas Koehler, and Herman Suit. "Proton Radiation as Boost Therapy for Localized Prostatic Carcinoma." In: *Jama* 241.18 (1979).

[SC95] Helmut Simonis and Trijntje Cornelissens. "Modelling Producer/Consumer Constraints." English. In: *Principles and Practice of Constraint Programming — CP '95*. Vol. 976. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 1995, pp. 449–462.

[SH11] Helmut Simonis and Tarik Hadzic. "A Resource Cost Aware Cumulative." In: *Recent Advances in Constraints*. Springer, 2011, pp. 76–89.

[Smi05] Barbara Smith. "Modelling for Constraint Programming." In: *Lecture Notes for the First International Summer School on Constraint Programming* (2005).

[Sol10] Christine Solnon. *Ant Colony Optimization and Constraint Programming*. Toronto, Canada: Wiley Online Library, 2010.

[Sun71] Yngve Sundblad. "The Ackermann Function. a Theoretical, Computational, and Formula Manipulative Study." In: *BIT Numerical Mathematics* 11.1 (1971), pp. 107–119.

[Tah+05] Djamel Nait Tahar, Farouk Yalaoui, Lionel Amodeo, and Chengbin Chu. "An Ant Colony System Minimizing Total Tardiness for Hybrid Job Shop Scheduling Problem with Sequence Dependent Setup Times and Release Dates." In: *Proceedings of the International Conference on Industrial Engineering and Systems Management*. Marrakech, Marocco, 2005, pp. 469–478.

[Tar75] Robert Endre Tarjan. "Efficiency of a Good but Not Linear Set Union Algorithm." In: *Journal of the ACM (JACM)* 22.2 (1975), pp. 215–225.

[Vil04a] 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*. Nice, France: Springer, 2004, pp. 335–347.

[Vil07] Petr Vilım. "Global Constraints in Scheduling." PhD thesis. PhD thesis, Charles University in Prague, Department of Theoretical Computer Science and Mathematical Logic, Prague, Czech Republic, 2007.

[VB02] Petr Vilım and Roman Barták. "Filtering Algorithms for Batch Processing with Sequence Dependent Setup Times." In: *Proceedings of the 6th International Conference on AI Planning and Scheduling, AIPS*. Vol. 2. Alberta, Canada, 2002, pp. 312–321.

[VBČ05] Petr Vilím, Roman Barták, and Ondřej Čepek. "Extension of O(nlog n) Filtering Algorithms for the Unary Resource Constraint to Optional Activities." In: *Constraints* 10.4 (2005), pp. 403–425.

[Vil04b] Petr Vilím. "O(nlogn) Filtering Algorithms for Unary Resource Constraint." English. In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Vol. 3011. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2004, pp. 335–347.

[Wil03] James Wilson. "Gantt Charts: A Centenary Appreciation." In: *European Journal of Operational Research* 149.2 (2003), pp. 430–437.

[Wol09] Armin Wolf. "Constraint-Based Task Scheduling with Sequence Dependent Setup Times, Time Windows and Breaks." In: *GI Jahrestagung* 154 (2009), pp. 3205–3219.

[WS04] Armin Wolf and Hans Schlenker. "Realizing the Alternative Resources Constraint Problem with Single Resource Constraints." In: *Proceedings of the INAP workshop*. 2004.

[WN14] Laurence Wolsey and George Nemhauser. *Integer and Combinatorial Optimization*. Wiley Series in Discrete Mathematics and Optimization. Wiley, 2014.

[WB06] Rolf Wüstenhagen and Michael Bilharz. "Green Energy Market Development in Germany: Effective Public Policy and Emerging Customer Demand." In: *Energy Policy* 34.13 (2006), pp. 1681 –1696.

[Zam+13] Stéphane Zampelli, Yannis Vergados, Rowan Van Schaeren, Wout Dullaert, and Birger Raa. "The Berth Allocation and Quay Crane Assignment Problem Using a CP Approach." In: *Principles and Practice of Constraint Programming*. Springer. 2013, pp. 880–896.

[ZP07] Alessandro Zanarini and Gilles Pesant. "Generalizations of the Global Cardinality Constraint for Hierarchical Resources." In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*. Springer, 2007, pp. 361–375.

[ZBT07] Eckart Zitzler, Dimo Brockhoff, and Lothar Thiele. "The Hypervolume Indicator Revisited: On the Design of Pareto compliant Indicators Via Weighted Integration." In: *Evolutionary Multi-Criterion Optimization*. Vol. 4403. Lecture Notes in Computer Science. Springer Berlin Heidelberg, 2007, pp. 862–876.