Sur le tri de tâches pour résoudre des problèmes d'ordonnancement avec la programmation par contraintes

Arthur Godet

DOCTORAT BRETAGNE LOIRE MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

THÈSE DE DOCTORAT DE

L'ÉCOLE NATIONALE SUPÉRIEURE
MINES-TÉLÉCOM ATLANTIQUE BRETAGNE
PAYS-DE-LA-LOIRE - IMT ATLANTIQUE

ÉCOLE DOCTORALE N° 601
Mathématiques et Sciences et Technologies de l'Information et de la Communication
Spécialité : *Informatique*

Par
Arthur GODET

Sur le tri de tâches pour résoudre des problèmes d'ordonnancement avec la programmation par contraintes

Thèse présentée et soutenue à IMT Atlantique – campus de Nantes – Amphi Kastler, le 23/09/2021
Unité de recherche : Laboratoire LS2N
Thèse N° : 2021 IMTA 0257

Rapporteurs avant soutenance :

Christian ARTIGUES	Directeur de recherche au LAAS-CNRS
Claude-Guy QUIMPER	Professeur à l'Université Laval

Composition du Jury :

Président :	Samir LOUDNI	Professeur à IMT Atlantique
Rapporteurs :	Christian ARTIGUES	Directeur de recherche au LAAS-CNRS
	Claude-Guy QUIMPER	Professeur à l'Université Laval
Examinateurs :	Hadrien CAMBAZARD	Maître de conférences à l'Université Grenoble Alpes
	Christine SOLNON	Professeure à l'INSA de Lyon
Dir. de thèse :	Nicolas BELDICEANU	Professeur à IMT Atlantique
Co-dir. de thèse :	Xavier LORCA	Directeur centre Génie Industriel à IMT Mines Albi-Carmaux
Encadrant de thèse :	Gilles SIMONIN	Maître de conférences à IMT Atlantique

Acknowledgements

First of all, I would like to thank Xavier Lorca, to introduce me early to the academic research environment. Among all the opportunities he offered me was this thesis, for which I also want to thank Nicolas Beldiceanu and Gilles Simonin for guiding me all along my research, for the discussions we had and for redirecting me when I got scattered by my ideas. It was a pleasure and an honour to have them as my supervisors and I learned a lot by their side.

I am honoured to have such an outstanding jury. I would like to thank Christian Artigues and Claude-Guy Quimper to have accepted to be my rapporteurs. Their insightful remarks, as well as the ones of Christine Solnon, helped me to improve on this manuscript, most especially by clarifying some ambiguities. I would like to thank Hadrien Cambazard, Samir Loudni and Christine Solnon to be assessors during the jury. I would like to thank Gilles Pesant and Christine Solnon for accompanying me all along the thesis. Finally, I want to thank Emmanuel Hébrard for the collaboration all along the thesis, most especially for his advice, the discussions we had on theory and practice of Constraint Programming and the knowledge he brought to redirect our research.

I would like to thank Charles and Philippe for all the morning coffee breaks, it was the best way to start any workday, whether the discussions were serious or not. I would especially like to thank Charles Prud'homme for his precious advice and help to better understand constraint solvers, most especially Choco solver. He contributed to this thesis much more than he suspects. I would like to thank the members and the former PhD students of the TASC team for all the good discussions, the coffee breaks, the lunch breaks and, of course, their good mood which makes every day a good day of work: Charles Prud'homme, Philippe David, Gilles Simonin, Charlotte Truchet, Samir Loudni, Nicolas Beldiceanu, Romuald Debruyne, Giovanni Lo Bianco, Pierre Talbot, Mathieu Vavrille, Ekaterina Arafailova, Gilles Madi-Wamba, Anicet Bart, Charles Vernerey, Jovial Cheukam Ngouonou. Of course, I also thank my officemates for all the good discussions: Giovanni Lo Bianco, Paula Metzker and Charles Vernerey.

My friends have a special place in my thanks. There have been ups and downs during these three years, and they have always been there to cheer me up and encourage me to continue to the end. Many thanks to your support and all the good moments. So, thank you: Adèle, Elisa, Vincent, Sofiane, Sarah, Alexis, Flora, Robin, Maxime, Guillaume, Romain, Elise, Loïc, Tristan, Guillaume, Marine, Léa, Joachim, Juliette, Yann, Kévin, Guillaume and Azel.

Finally, I would like to thank my family, and most especially my parents Nathalie and Hervé, and my sister Amandine, for their unconditional support, no matter how distant I might have been during some periods due to my work. I cannot thank you enough for everything you did and do for me.

Contents

List of Figures

List of Tables

List of Algorithms

To my parents

Introduction

Efficiency in industrial processes, transportation and society in general is important, today more than ever. Indeed, our modern societies have scaled up, as well as their underlying complexity and entanglement. For instance, logistics and transportation play a key role in the functioning of our societies. Optimising the underlying processes, as well as every activity indispensable to services, is therefore important, most especially in the context of climate change and societal transition. Indeed, energy and money savings are important to preserve the quality of life of every citizen as much as possible in such a context.

Operations Research precisely consists in modelling a problem with mathematical variables and constraints and to solve the modelled problem. Operations Research concentrates on the practical resolution of problems and therefore uses solving techniques from several fields. Different techniques exist to solve these problems, some easier to implement in code than others, some being able to guarantee the nonexistence of solutions or the optimality of a solution, some being faster in practice than others, and so on. The diversity of solving techniques allows for selecting the appropriate one depending on the project constraints (skills, time for developments, time to find a solution, ability to prove optimality or not, etc.).

Problems whose objective is to find a solution are generally called Constraint Satisfaction Problems (CSP) and Constraint Optimisation Problems (COP) whenever there is an objective-function to minimise or maximise in addition of the CSP. These problems have in common to be combinatorial, that is the set of possibilities is very large. Various types of problems are modelled as CSP and COP. Just to cite a few, it goes from maintenance scheduling in the electricity industry [Fro+16], to logistics transportation and delivery problems (generally called Vehicle Routing Problems [GRW08; KP12; MS20]), or building the timetable of trains in stations [Bai15; CGT15] or even the conception of the new French keyboard [JK19; Fei+21] (modelled as a specialisation of the Quadratic Assignment Problem [BK57]).

In the context of this thesis, we concentrate on a solving technique called Constraint Programming (CP), which is a declarative paradigm, that is the user specifies the problem into a constraint language, interpretable by a software called a constraint solver, whose task is to solve the problem. This "black box" behaviour exists from the beginning of the field and in the mind of researchers: "Constraint programming represents one of the closest approaches computer science has yet made to the Holy Grail of programming: the user states the problem, the computer solves it" as stated Freuder in [Fre97]. Indeed, CP distinguishes from other techniques such as Linear Programming by the great expressivity it allows for constraints, as well as in the possibility to integrate into constraint solvers algorithms and techniques from other fields. Sadly, this great expressivity and permissiveness of CP leads to complex constraint solvers, which require a certain expertise in order to efficiently model and express a problem into and efficiently solve it. We can call this behaviour a "white box": the solver allows many things, but it requires expertise to configure it efficiently. In this thesis, we will be closer to the "white box" behaviour than the "black box" one.

Context and contributions

To solve problems in CP, one first needs to model the problem at stake by the means of variables and constraints. For instance, let us say that you want to schedule the maintenance of a fleet of trains. You have a set of tasks, each one corresponding to a maintenance activity. The objective of your problem is to decide when to process each of the task, given resource constraints (two trains cannot occupy the same rail at the same time, timetable of the skilled workers, etc.) and planning constraints (the trains arrive and leave the maintenance centre at specific time to respect their commercial schedule for instance). Considering that tasks have a fixed duration to be processed, the start variables of the tasks define a solution of the problem if none of the constraints is violated. The modeller can also specify how the search space will be explored: we consider it part of the modelling phase. Once the modelling phase is completed, the solver starts exploring the search space, looking for solutions. All the variable-value assignments defined in this phase constitute the search space.

The solver alternates between two phases: propagation and branching. The role of propagation is to infer that some variable-value assignments cannot lead to a solution of the problem given the definitions of the constraints. For instance, if it was already decided that the electricity maintenance of a given train would be done from t_1 to t_2 and that there are machines to do electricity maintenance only for one train at a time, then the propagation engine can enforce that other electricity maintenance tasks of other trains ends before t_1 or starts after t_2. This propagation phase can lead to great cuts into the search space and should therefore not be underestimated. However, these inferences have a certain algorithmic cost, generally the stronger the inference, the costly it is to find it. In this thesis, we will give attention to this interesting trade-off.

Whenever the inference rules of constraints cannot find failing variable-value assignments anymore, either all variables are instantiated and a solution has been found, either there are contradictions between constraints inferences and we should go back to a previous valid state of the search, either it is undecided between the two former. In this third case, a branching is done, *i.e.* the solver takes a decision (for instance electricity maintenance of train A will be done at time t_1) and propagates this decision. The contradiction of this decision will also be explored. This is why Constraint Programming is categorised as a *complete* search method: it explores all possibilities and therefore can guarantee the nonexistence of solutions or the optimality of a solution (for example, it can guarantee that no valid schedule taking less total time can be found). Choosing the good decision to branch on first can be crucial for performance.

As a matter of fact, in his speech during the ACP Research Excellence Award (CP'13)[1], Jean-Charles Régin explains that paying attention to the modelling phase, to the exploration of the search space and to the implementation (especially to data structures) is important, the potential gain in efficiency following this order and being inverse to the probability of success. That is, among these three phases, the highest gain of efficiency can be found in the modelling phase, but there is in fact few chances of actually getting an efficiency improvement.

Another point of attention, despite not really in the hands of modellers and solvers' users, is the propagation engine: how it is implemented and articulated might have a great importance in practice when solving problems, as was shown by Prud'Homme

[1] "Simple solutions for complex problems", Jean-Charles Régin. ACP Research Excellence Award, CP2013(Uppsala, Suède).

et al. [Pru14; Pru+14]. More especially, great knowledge of a solver and its inner functioning can lead to greater performance in practice as the implementation will be adapted to the solver's specifics. For all these reasons, we will give particular attention to implementation details all along this thesis.

In this thesis, we will explore different aspects presented earlier. We mainly focus on improving the practical performance of solving scheduling problems. More especially, we exploit the great expressivity of CP to introduce into constraint solvers other techniques from Combinatorial Optimisation. Doing so has proven to be very efficient in the past: the use of explanations (a technique coming from SAT solvers) [Jus03; Sch+09; Sch+11; SFS13] have led to impressive results on classical scheduling problems, and is a strong asset of CP to efficiently solve scheduling problems. Sadly, it takes a lot of code development efforts to interface explanations with the propagation engine of the solver (as was done in Choco [PFL17]) or to natively take them into account (such as in lazy-clause generation (LCG) solvers such as Chuffed [Chu+16] or OR-Tools [PF], also called CP-SAT solvers). Also, explanations can face difficulties: for instance, the explained decomposed version of the CUMULATIVE constraint [AB93] can consume a very large amount of memory if the schedule can be built over a large time period. This excludes the use of explanations in memory-limited environments, where explanations might lose in efficiency as some clauses are forgotten (which is done by LCG solvers in memory-limited environments). For these reasons, we concentrate on other techniques, which are complementary with explanations.

The contributions of this thesis are twofold. First, we revisit the ALLDIFFPREC constraint, which was introduced by Bessiere et al. [Bes+11] as an ALLDIFFERENT constraint with precedence constraints between variables. We present in details all state-of-the-art results on this constraint, and show how to correct the existing filtering algorithm. We also give another filtering algorithm based on the same idea but whose worst-case time complexity is different. Then, we present our main contribution on this constraint, which is a stronger filtering algorithm that considers variables' domains as sets instead of intervals. We will experimentally show that, despite its high worst-case time complexity, our new filtering rule can have great effects in practice. We will especially analyse the behaviour of each filtering algorithm depending on the density of the domains.
On a second hand, we explore on the use of list ordering algorithms in constraint solvers. List ordering algorithms are problem-specific algorithms that build a solution from a list of the variables. We introduce a new generic constraint, called ORDER, that embeds the inner reasoning of the list ordering algorithm and show how models can be improved using additional variables and constraints, most especially the ORDER constraint. We will apply this methodology on several scheduling problems, showing how to implement the ORDER constraint for these problems, and we will compare experimentally the new resulting model with the former one. We will stress the resulting propagators and application cases to show potential limits of the methodology.

Organisation of the thesis

The first part of this thesis introduces all the notions that are needed to deeply understand our research. Chapter 1 gives the key fundamentals on algorithms' complexity and theoretical computer models that will be used all along the thesis. Chapter 2 presents Constraint Programming, giving details to understand the different phases

of CP, *i.e.* modelling, propagation and branching, and how each of these phases work. As we said, CP allows a great expressivity and diversity of solving techniques. We will therefore specify in Chapter 2 which ones are used in the context of these researches. Finally, Chapter 3 present in details the scheduling problems on which the Order constraint will be applied on, as well as minimal CP models for these problems.

In a second part, we revisit the allDiffPrec constraint. In Chapter 4, we concentrate on the definition and presentation of each filtering algorithm, including our new stronger one. A deep analysis compares the different schemes in terms of filtering strength as well as worst-case time complexity. In Chapter 5, we discuss on the implementation details of each filtering scheme. More especially, we discuss on the data structures used in the algorithms and how they should be implemented to be efficient in practice. Different implementations are also discussed for the greedy filtering scheme. Finally, the experimental protocol and results are presented and analyse.

The third part is dedicated to the introduction of list ordering reasoning into CP, through the new Order constraint. Chapter 6 presents the generic Order constraint and a basic filtering scheme for it. Generic notations and notions are given, showing what needs to be concretely defined in applications of the constraint. Chapter 7 presents our results for the Parallel Machine Scheduling Problem with Additional Unit Resources (PMSPAUR), most of which were published in [God+20]. Application of the Order constraint, in the form of the leftShifted constraint, to the Resource-Constrained Project Scheduling Problem (RCPSP) is given in Chapter 8. In Chapter 9, we show how to specify the Order constraint to the Unit Execution Time-Unit Communication Time (UET-UCT) problem where duplication of tasks are authorised.

Finally, we will summarise the contributions of this thesis and gives insight on further research that can be lead on the subjects treated in this thesis.

Part I

State of the art

Chapter 1

Complexity and Graph Theory

In this chapter, basic notions of complexity theory in Computer Science will be defined. More especially will be discussed how algorithms' theoretical execution speed is defined and how it is used to compare algorithms. On a second part of this chapter, some notions of graph theory will be defined. All the notions defined in this chapter will either be used within next chapters or help painting a broad picture of the field.

1.1 Basic notions of Complexity Theory

1.1.1 The Bachmann-Landau notation

First, let us remind some mathematical notions that were introduced by Bachmann [Bac94] and adopted and extended by Landau [Lan09]. Let f be a mathematical function defined over reals $\mathbb{R}$. We consider a *strictly positive* function g over reals $\mathbb{R}$. The idea is for g to represent the *asymptotic* behaviour of function f on some part of the real set $\mathbb{R}$.

Definition 1.1 (Bachmann-Landau notation)**.** *Let $a \in \mathbb{R}$ be a real. The* Bachmann-Landau notations *or* big O notations *are defined as following:*

- $f(x) \underset{x \to \infty}{=} O(g(x))$ *iff* $\exists x_0 \in \mathbb{R}, M > 0, \forall x \geq x_0, |f(x)| \leq Mg(x)$
- $f(x) \underset{x \to a}{=} O(g(x))$ *iff* $\exists M, \epsilon > 0, \forall x \in [a - \epsilon, a + \epsilon], |f(x)| \leq Mg(x)$

The first Bachmann-Landau notation can be extended to the set of integers $\mathbb{N}$. In such a case, functions f and g are defined over $\mathbb{N}$ and their values are still in $\mathbb{R}$ for f and in $\mathbb{R}^+$ for g. In Computer Science, only the first notation will be used, that is when $x \to \infty$. As such, we will note in the remainder of this thesis $f(n) = O(g(n))$ instead of $f(n) \underset{n \to \infty}{=} O(g(n))$.

Even if the function g can have various forms, some are very common. Whenever g is a sum of sub-functions, only the ones with the largest growth rate are kept in the final notation, the rest being omitted. For instance, even if $g(n) = n^2 + 3n + 2$ is more detailed, only n^2 is considered important to characterise the asymptotic behaviour of f for a big O notation. Constant coefficients are also removed from g function because it does not change the asymptotic behaviour, nor the definition. Whenever g is a polynomial of degree k, we will therefore note $O(n^k)$.

Finally, let us remark that in the Bachmann-Landau notation the use of the $=$ sign is an abuse of notation: the notation is not reversible. For example, $n = O(n^2)$ but the reverse is not true.

1.1.2 On computability and decidability

In the 19^{th} century, Italian mathematician Giuseppe Peano wrote down the 8 first-order axioms on natural numbers and arithmetic. The idea behind this clarification was to build the fundamentals of arithmetic and understand if all formulas can be derived and verified only using these axioms, that is the *formal system* is *complete*.

At the beginning of the 20^{th} century, mathematicians work a lot on formal systems, and more especially on the notion of *decidability*. In mathematical logic, it is said of a set $\mathcal{T}$ that it is *closed* if it contains every formula ϕ that can be logically deduced from $\mathcal{T}$ (using the axioms of the formal system). A *logical consequence* is a combination of uses of authorised manipulations of the formal system's symbols and axioms from one true statement to another. A *theory* $\mathcal{T}$, that is a set of sentences closed under logical consequence, is *decidable* if and only if there exists an effective method for determining if a formula is in the theory. A formal system is *complete* if any formula or its negation can be logically proved from the axioms. It is *consistent* if it is not possible to prove a formula and its negation from the axioms. In 1931, Kurt Gödel showed that any consistent formal system embedding Peano's axioms of arithmetic is incomplete, that is there exist some statements that can be neither proved nor disproved [Göd31].

On a more general context, Hilbert and Ackermann asks in 1928 whether it is possible, from a given set of symbols and axioms defining authorised operations and relations on it, to find an effective method to determine from the axioms if a given statement is true of false: the Entscheidungsproblem [HA28]. In 1936, Alonzo Church [Chu36b; Chu36a] and Alan Turing [Tur36] both proved that it is impossible to find a general solution to the Entscheidungsproblem, the former using lambda calculus and the latter using Turing machines (both being proven Turing-equivalent by Turing).

In the remainder of this subsection, we will concentrate on a more formal definition of the notions needed in the next subsection on analysis of algorithms. The precedent paragraphs were only for historical purpose, but it is interesting to see where these notions came from, as it might not been that obvious nowadays.

Even if there exist other types of Turing machines, we will concentrate here on *1-tape deterministic Turing machine*, which we will call Turing machine without former ambiguity. A Turing machine is an abstract machine consisting of an infinite tape divided into cells. Each cell contains either a symbol from a given *alphabet* either a *blank* symbol. This tape is scanned by a *tape head* which is able to read and write symbols from the alphabet into the cell it is currently over. The machine has a set of authorised *states*. A *move* of the head consists of:

1. An update of the state (possibly the same),
2. An update of the symbol in the scanned cell (possibly the same),
3. And a move of the tape head to the direct cell on the left or the right.

Initially, the tape head is at the far left of the input and in an initial state. Finally, a *transition function* defines what the machine does, that is how the tape head moves, given a state and a cell as inputs.

A Turing machine *halts* whenever it is in a final state.

Definition 1.2 (Computable function [Tur36])**.** *Considering the model of computation of the Turing machines, a function* $f : \mathbb{N}^k \to \mathbb{N}$ *is* computable *if and only if there exists a Turing machine that terminates while computing* $f(x)$ *with* $x \in \mathbb{N}^k$*.*

Definition 1.3 (Decidability)**.** *Considering a formal system and its axioms and symbols, a statement is* decidable *if and only if there exists a Turing machine, with the statement as input and whose transition function is defined only using the formal system's axioms, that terminates with a "true" or "false" answer for the statement. It is* undecidable *otherwise.*

The halting problem consists in answering, for a given input and Turing machine, if the Turing machine will halt or run forever. In [Tur36], Turing shows that the Entscheidungsproblem is undecidable by showing that there does not exist a general Turing machine solving the halting problem for all possible program-input pairs.

Definition 1.4 (Turing completeness and equivalence)**.** *A model of computation is* Turing complete *if it can compute all computable functions.*

A model of computation is Turing-equivalent *if it can simulate a Turing machine and if a Turing machine can simulate it.*

1.1.3 Analysis of algorithms

An *algorithm* is defined by a finite sequence of logical and well-defined instructions to solve a given problem (sorting integers, finding the maximum number within a list, computing the i^{th} decimal place of π, etc.). Such instructions can be assigning a value to a variable, adding two variables, and so on. For its good execution, an algorithm needs some resources, more especially time and memory space. Therefore, estimations of these resources that the algorithm is going to consume upon its *run-time* is very interesting for analysing and comparing algorithms as well as to anticipate their execution. In Computer Science, such estimations are commonly called the *space complexity*, *i.e.* the estimation of maximum memory needed to be allowed to the algorithm for its good execution, and the *time complexity*, *i.e.* the estimation of the run-time of the algorithm. Given that, nowadays, memory is cheap and easy to add to most computers, intensive focus is done on the time complexity. Space complexity is still interesting to keep in mind (see Remark 1.1), and can even be essential in a space-limited environment as embarked robotics, satellites, etc. Therefore, from here, we might simplify "time complexity" into "complexity". If there is a risk of misunderstanding, we will precise which type of complexity is discussed about.

Remark 1.1. *Despite memory being cheap and easy to add to most computers, memory consumption should still be in every developer's mind. Indeed, every computer's memory is limited and thus all software cannot consume more and more memory on their own. This is among the main reasons of computer replacements: slowdown of the system because of the lack of memory. When all software are using lots of memory, the entire computer system tends to get slower. Therefore,* software ecodesign *encourages developers to optimise the memory-consumption of the software they develop as well as the run-time.*

The time complexity of a given algorithm is not given in terms of time per se, but rather as a function of the inputs' size whose result is the number of basic instructions, also called *unit-cost operations*. As an example of a time complexity expressed with more than one input, the Hopcroft-Karp algorithm [HK73], that will be seen later in this chapter, has a time complexity of $O(m\sqrt{n})$ with m the number of edges and n the number of nodes (these notions will be seen with more details in Section 1.2). The unit-cost operations are different from one *model of computation* to another. Two very common models used in analysis of algorithms are *Turing machines* and *random-access memory machines* (or *RAM machines*) which was first described by Cook and

Reckhow [CR73]. Turing machines are more of a theoretical interest, while RAM machines are closer to the functioning of real-life computers. Unit-cost operations, as listed in [AHU74], are: LOAD, STORE, ADD, SUB, MULT, DIV, READ, WRITE (8 operand operations), JUMP, JGTZ, JZERO (3 label operations) and HALT. These 12 unit-cost operations allow to describe all algorithms within the RAM machines model of computation. We will not go into details for each of these operations and will just say that any basic mathematical operation (addition, subtraction, multiplication and division) between values are unit-cost operations as well as reading and writing values within the memory.

Currently, there are two main cost models commonly admitted and used to evaluate the time complexity of an algorithm, both formally described in 1974 by Aho et al. [AHU74]. The *logarithmic cost model* offers more details as it distinguishes the cost of every unit-cost operations. In fact, the logarithmic cost model details the cost proportionally to the number of bits used during the operation. However, such a detailed approach takes time to compute an algorithm's time complexity. It is critical in some contexts, such as cryptography, but common analyses of algorithms do not need as many details. The *uniform cost model* considers that all unit-cost operations have the same cost: 1 unit of time. Even if all the operations do not take the same amount of time in practice (a multiplication could be 100 times slower than an addition), this is a realistic model for most use cases in Computer Science.

In the remainder of this thesis, all time complexities of algorithms will be expressed with the hypothesis of a *uniform cost model for RAM machines* in mind as it is Turing-equivalent [CR73; HPV75].

Now that we have a model of computation, we can compute the time complexity of an algorithm. For this, we associate to each instruction the number of unit-cost operations it needs to be correctly executed and multiply it with the number of times the instruction is executed during the algorithm's run-time. The time complexity of the algorithm is the sum of the cost of all instructions, *i.e.* the total number of unit-cost operations that were executed along the run-time of the algorithm.

Given the input's size n, several type of analyses can be done on algorithms. The two most used are the *average-case complexity* and the *worst-case complexity*. The *average-case complexity* focuses on the complexity of the algorithm to solve the problem in average, that is the complexity is defined with respect to a probabilistic distribution of the inputs. The *worst-case complexity*, which is probably the most used type of algorithms analyses, gives the complexity of the algorithm for the worst input of a given size n. For the worst-case complexity, it is common to compute an upper bound of the real worst-case complexity. However, one should be careful when computing the complexity as an algorithm with $O(n \log(n))$ complexity is also an $O(n^2)$ algorithm, but when looking for the better algorithm to solve a given problem, one is inclined to select the faster algorithm, that is the one with the smallest time complexity.

Finally, we would remind that algorithms' complexities are theoretical tools. However, the performance estimated by these tools is not necessarily a good reflection of the performance measured in practice. Let us take, as an example, the problem of sorting a list of n integers into increasing order. Among the most efficient known algorithms for this problem, there are the `merge sort` algorithm [NT63] and the `quicksort` algorithm [Hoa61]. The former has an average-case complexity of $O(n \log(n))$ and a worst-case complexity of $O(n \log(n))$, while the latter has an average-case complexity of $O(n \log(n))$ and a worst-case complexity of $O(n^2)$. One could be inclined to choose the merge sort algorithm as its complexities in average and worst

cases are better or equivalent to the complexities of the quicksort algorithm. However, it has been shown that the quicksort algorithm is faster in practice than the merge sort algorithm in certain situations (type data structure used for the list, etc.). When selecting an algorithm to solve a given problem, one should therefore not necessarily rely only on the algorithms' complexities as algorithms with worse complexities can be in fact faster in practice than algorithms with better complexities.

1.1.4 On complexity classes

From the complexities computed with the notions seen in the previous subsection, problems have been grouped into families and the complexities have been classified depending on how high they are.

Definition 1.5 (Polynomial complexity)**.** *An algorithm, whose worst-case complexity can be written as $f(n)$, has a* polynomial complexity *if and only if its complexity is bound by a polynomial of order k ($k \in \mathbb{N}$ can be as high as necessary), i.e. $\exists k \in \mathbb{N}, f(n) \leq n^k$. We also say that such an algorithm is polynomial.*

A *decision problem* is a mathematical problem whose expected solution is either "yes" or "no". For instance, knowing if it is possible to go from Nantes, France to Lille, France by car in less than 700km is a decision problem.

Definition 1.6 (Class $\mathcal{P}$)**.** *$\mathcal{P}$ is a class of complexity that contains all the decision problems that are decidable and for which decidability can be proven with a polynomial algorithm.*

Definition 1.7 (Class $\mathcal{NP}$)**.** *$\mathcal{NP}$ is a class of complexity that contains all the decision problems for which a solution can be checked with a polynomial algorithm, that is there is a polynomial algorithm that attests if a given assignment of variables is a valid solution of the problem or not.*

It is obvious that $\mathcal{P} \subseteq \mathcal{NP}$. The question is still open whether $\mathcal{P} = \mathcal{NP}$ or $\mathcal{P} \neq \mathcal{NP}$, notwithstanding it is assumed by most mathematicians and computer scientists that $\mathcal{P} \neq \mathcal{NP}$ (most results are built upon this hypothesis).

It is common to distinguish a problem from its *instances*. A problem is then a general mathematical problem based on fixed but unknown data, whereas an instance of a problem has the same definition of the problem but the data are known. Throughout this thesis, we will be meticulous to talk about *problems* for the general mathematical form and of *instance* for a form of a given problem with specified data. Given a decision problem Π, we note $\mathcal{I}_\Pi$ the set of its instances.

In the theory of complexity, as we can already see, all problems are not as easy to solve and a hierarchy can be done. To help with this, reductions are powerful tools. The most known and used one is the *Turing reduction*.

Definition 1.8 (Turing reduction)**.** *Let Π and Π' be two decision problems. A* Turing reduction*, noted $\leq_T$, reducing problem Π to problem Π' is a function $f : \mathcal{I}_\Pi \to \mathcal{I}_{\Pi'}$ such that:*

- *f is computable in polynomial time*
- *Given a function $g(P, I_P)$ answering yes or no to the instance I_P of problem P, then: $\forall I_\Pi \in \mathcal{I}_\Pi,\ g(\Pi, I_\Pi) \iff g(\Pi', f(I_\Pi))$*

Proposition 1.1 (Transitivity of Turing reduction)**.** *Turing reductions are* transitive*, that is if* $\Pi_1 \leq_T \Pi_2$ *and* $\Pi_2 \leq_T \Pi_3$*, then* $\Pi_1 \leq_T \Pi_3$*.*

Definition 1.9 ($\mathcal{NP}$-completeness)**.** *A decision problem* Π *is* $\mathcal{NP}$-complete *if and only if:*

- $\Pi \in \mathcal{NP}$
- $\forall\ \Pi' \in \mathcal{NP}, \Pi' \leq_T \Pi$

The notion of $\mathcal{NP}$-completeness was introduced by Cook [Coo71] when he proved that the SATISFIABILITY (or SAT) problem was $\mathcal{NP}$-complete. His work was extended by Karp in [Kar72] and his "21 $\mathcal{NP}$-complete problems". Given existing $\mathcal{NP}$-complete problems, Lemma 1.1 shows how to prove that a given problem is $\mathcal{NP}$-complete.

Lemma 1.1. *Let* Π *be a decision problem. To show that* Π *is* $\mathcal{NP}$*-complete, it is sufficient to show that:*

- $\Pi \in \mathcal{NP}$
- $\exists\ \Pi'$ $\mathcal{NP}$*-complete such that* $\Pi' \leq_T \Pi$

Definition 1.10 ($\mathcal{NP}$-hardness)**.** *A decision problem* Π *is* $\mathcal{NP}$-hard *if and only if it can be reduced to any* $\mathcal{NP}$*-complete problem:* $\exists\ \Pi'$ $\mathcal{NP}$*-complete,* $\Pi' \leq_T \Pi$*. Thus, having an oracle for a* $\mathcal{NP}$*-hard problem, any* $\mathcal{NP}$*-complete problem can be solved in polynomial time.*

$\mathcal{NP}$-hard problems can be seen as problems that are "at least as hard" as $\mathcal{NP}$ problems to solve.

$\mathcal{NP}$-complete problems are problems that are in $\mathcal{NP}$ and that are $\mathcal{NP}$-hard.

Garey and Johnson gave an extensive survey on the notion of $\mathcal{NP}$-hardness in [GJ79]. This reference book gives a list of known $\mathcal{NP}$-complete and $\mathcal{NP}$-hard problems as well as all notions needed for these theories.

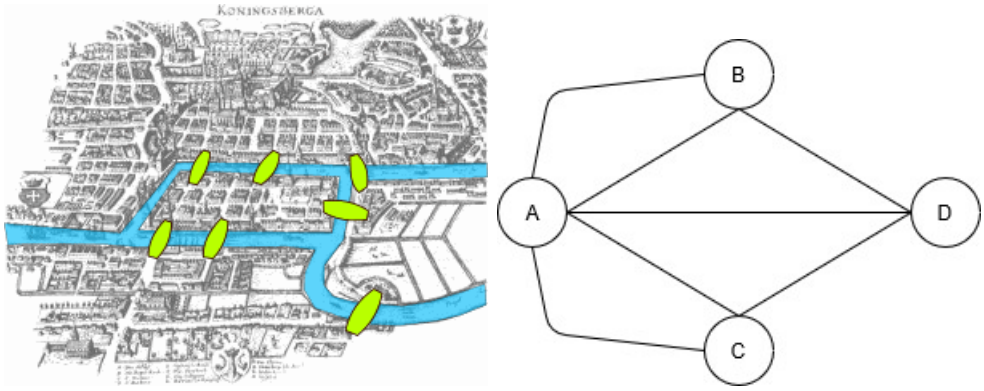
Decision problems are not the only kind of problems that computer scientists aim to solve. Among them are also *solution problems* which are similar to decision problems in such matter that they ask to exhibit the solution that is used to answer the decision problem. *Optimisation problems* are problems that aim to minimise a given quantity. For instance, "What is the minimal distance to visit all cities in France?" (which is an instance of the famous Travelling Salesman Problem) is an optimisation problem. Solution problems and optimisation problems have their given classes of complexity (see [Pap94] for details), which are similar to the ones of decision problems, such that it is common to amalgamate them in the names of the complexity classes of decision problems. We are well aware of such a notation's abuse but will do so in this thesis for commodity.

1.2 Basic notions of Graph Theory

In this section, we will present notions of Graph Theory that will be of use in coming chapters. These notions mostly come from [GM95], that curious readers can look at for further information.

1.2.1 Introduction to Graph Theory with the problem of the seven bridges of Königsberg

Graphs are mathematical structures that present a wide variety of use, both theoretically and practically. The foundation of Graph Theory can be traced back to 1736 with a paper of Leonhard Euler [Eul36]. Königsberg was set on both sides of the Pregel River. Two islands were connected to each other or to both parts of the mainland by a total of seven bridges (see Figure 1.1a). Euler discussed the possibility to cross all seven bridges during a walk without crossing twice the same bridge, which is now known as "The problem of the seven bridges of Königsberg". To answer such a question, he introduced a mathematical structure, represented in Figure 1.1b, that is called nowadays a *graph*. A graph is composed of *vertices* (also called *nodes*) and of *edges*, which are links between two vertices. In Figure 1.1b, each edge represents a bridge, while vertices represent either an island or a mainland. Vertex A represents the left island, vertex D the right island, while vertex B represents the upper mainland and vertex C represents the lower mainland.



(A) Map that inspired the problem[1] (B) Graph representation of the problem of the seven bridges of Königsberg

FIGURE 1.1: The problem of the seven bridges of Königsberg

The order n of a graph is the total number of vertices. The *size* m of a graph is the number of edges. As the edges are not oriented, the graph is said to be *undirected* (it is *directed* otherwise). If an edge is oriented, it can be used only in the given direction, in which case we talk about *arcs* instead of oriented edges. Formally, a graph is noted $G = (V, E)$ with V being the set of vertices and E the set of edges. An edge between vertices v and w is noted (v, w). In an undirected graph, we suppose that $(v, w) \in E$ and $(w, v) \in E$. In a directed graph, $(v, w) \in E$ if and only if there is an arc from v to w. If $(v, w) \in E$ and $(w, v) \notin E$, there is an arc from v to w but not from w to v. The *degree* of a vertex is the number of incident edges. Two edges with a common vertex are said to be *adjacent*.

A finite sequence of edges of E is a *walk*. If all edges are distinct, it is a *trail*. A *path* is a trail where all the vertices are distinct. Whenever the first and last vertices of a path are the same, the path is characterised as a *cycle*. If the graph does not have any cycle, it is *acyclic*. If, for any two different vertices $v, w \in V$, there exists a path from v to w, the graph is *connected*. If this property is true in a directed graph, the graph is *strongly connected*. $G' = (V', E')$ is a *subgraph* of $G = (V, E)$ if and

[1] This image, made by Bogdan Giuşcă, is distributed under Creative Commons Attribution-Share Alike 3.0 Unported License.

only if $V' \subseteq V$, $E' \subseteq E$ and $\forall (v,w) \in E', v \in V'$ and $w \in V'$. A *strongly connected component* of a graph G is a maximal strongly connected subgraph of G, that is the strongly connected component cannot be extended with other nodes of G without violating the strongly connected property. In [Tar72], Tarjan gives an algorithm to find all connected components of a graph in linear time. This algorithm also gives all the *articulation points*, that is the vertices such that their removal would increase the number of connected components.

The *neighbourhood* $N(v)$ of a vertex v in an undirected graph $G = (V, E)$ is the set of vertices $w \in V$ such that $(v, w) \in E$: $N(v) = \{w \in V \mid (v, w) \in E\}$. In a directed graph, the *predecessors* of a vertex v are its neighbours $w \in V$ such that $(w, v) \in E$ and its *successors* are its neighbours $z \in V$ such that $(v, z) \in E$. The set of predecessors is noted $N^-(v)$ while the set of successors is noted $N^+(v)$.

For the graph representation of the problem of the seven bridges of Königsberg (Figure 1.1b), the graph has an order of 4, a size of 7 and is connected. Euler proved that the problem of the seven bridges of Königsberg cannot be solved: he showed that such a walk can exist only if there are exactly zero or two nodes of odd degree (it was later showed that it is also a sufficient condition).

A *forest* is an acyclic graph. A *tree* is a particular type of graphs $G = (V, E)$ that are connected and acyclic. They have the particular property that $|E| = |v| - 1$. A tree is generally represented with a node at the top, the *root node*, and a recursive form: each node is linked to a node in top of itself (its *parent node*) and a number of nodes below itself (its *children nodes*). A *binary tree* is a tree such that nodes have at most 2 children nodes. Nodes that don't have any children nodes are called *leaves*, other ones being called *internal nodes*. A *branch* of a tree starts from an edge of a parent node to a leaf node.

A *bipartite graph* is a graph $G = (V, E)$ such that vertices can be divided into two disjoint sets V' and V'' such that edges are necessarily from one set to the other, *i.e.* $\forall (v,w) \in E, v \in V' \wedge w \in V'' \vee v \in V'' \wedge w \in V'$. A bipartite graph is usually noted $G = (U, V, E)$ with U and V the two *parts*, *i.e.* the two disjoint sets, and E the set of edges. In such a notation, the order of the graph is $n = |U| + |V|$.

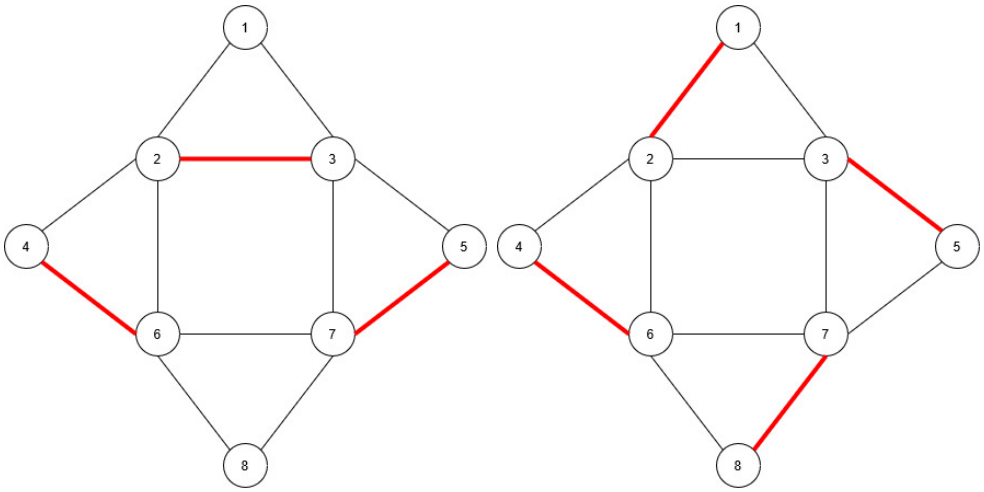
1.2.2 Matching theory

Definition 1.11 (Matching)**.** *Given a graph $G = (V, E)$, a* matching *M is a subset of the edges such that no two edges in M share a vertex. Edges in M are said to be* matched*, and* unmatched *otherwise. A vertex is* matched *if it is an endpoint of a matched edge, and* unmatched *otherwise. The* size *of a matching M, noted $|M|$, is the number of matched edges.*

Definition 1.12 (Maximal matching)**.** *A matching M of a graph $G = (V, E)$ is* maximal *if and only if there is no matching M' of G that contains M. In other words, every edge in E is either matched, either adjacent to a matched edge.*

Definition 1.13 (Maximum matching)**.** *A matching M of a graph $G = (V, E)$ is* maximum *if and only if there is no matching of G of greater size, i.e. M is maximum iff $|M| = \max_{M' \text{ is a matching of } G} |M'|$.*

Every maximum matching is of course maximal, but the converse is not true. Figure 1.2 shows two maximal matchings of the same graph, one of them being maximum.



(A) Maximal matching (B) Maximum matching

FIGURE 1.2: Maximal and maximum matchings

Definition 1.14 (Perfect matching)**.** *A matching M of a graph $G = (V, E)$ is* perfect *if and only if all the vertices are matched.*

Research have been led to found a maximum matching in a graph. Let us see how.

Definition 1.15 (Alternating path)**.** *Given a matching M of a graph $G = (V, E)$, an* alternating path *with M is a path that starts with an unmatched vertex and such that visited edges are alternatively unmatched and matched.*

Definition 1.16 (Augmenting path)**.** *Given a matching M of a graph $G = (V, E)$, an* augmenting path *with M is an alternating path with M that starts and finishes on unmatched vertices. Such vertices are also said to be* free*.*

Lemma 1.2 (Berge's Lemma [Ber57])**.** *A matching M of a graph $G = (V, E)$ is maximum iff there is no augmenting path with M.*

From Berge's Lemma (Lemma 1.2), it is possible to compute a maximum matching. We first start with a given matching M (that can be obtained by randomly selecting edges that are unmatched and that are not adjacent to matched edges until none are available) of size k. If an augmenting path P is found, the matching can be extended by removing all matched edges of P from M and by adding all unmatched edges of P to M. We now have a matching M of size $k + 1$. Following Berge's Lemma, we can repeat this process until no augmenting path can be found anymore. The resulting matching M is then maximum.

For a time, the Hungarian algorithm [Kuh55] was used to find a maximum matching in bipartite graphs. Originally with a time complexity of $O(|V|^4)$ [Kuh55], the complexity was improved to $O(|V|^3)$ by Edmonds and Karp [EK72] and by Tomizawa [Tom71]. In term of space complexity, the Hungarian algorithm has a worst-case complexity of $O(|V|^2)$.

In 1973, Hopcroft and Karp came with a better algorithm for finding maximum matching in bipartite graphs [HK73]. This algorithm, designed for bipartite graphs, has a worst-case time complexity of $O(m\sqrt{n})$ and a worst-case space complexity of $O(n)$, with n the number of vertices and m the number of edges. As $m \leq n^2$, one can easily see the gain in both time and space complexities, despite both algorithms being based on Berge's Lemma.

Algorithm 1.1: Hopcroft-Karp algorithm

```
1  Function FindMaximumMatching(G = (U, V, E) : bipartite graph) : integer
2  begin
3      for i ∈ U ∪ V do
4          matched[i] ← −1
5      end
6      F ← U ∪ V
7      do
8          found ← false
9          for u ∈ F ∩ U do
10             mate ← BFS(G, u, F)
11             if mate ≠ −1 then
12                 F ← F \ {u, mate}
13                 j ← mate
14                 do
15                     k ← next[j]
16                     matched[j] ← k
17                     matched[k] ← j
18                     j ← next[k]
19                 while k ≠ u
20                 found ← true
21             end
22         end
23     while found
24     return |{i ∈ U ∪ V | matched[i] ≠ −1}|/2
25 end
```

To implement the Hopcroft-Karp algorithm for a bipartite graph $G = (U, V, E)$, we consider without loss of generality that nodes in U are numbered from 0 to $|U| - 1$ and nodes in V are numbered from $|U|$ to $|U| + |V| - 1$. For convenience in notation, we will use the nodes' notation as their corresponding integer value. We consider an array of integers $next[]$ of size $|U| + |V|$ that will be used to represent the augmented path, should one exists, with $next[i] = j$ meaning that node j is after node i in the path. F is the set of free nodes, that is nodes that are still unmatched. Q is a queue, and we will note $push()$ (respectively $pop()$) the function to add a value at the end of the queue (respectively remove a value at the beginning of the queue). in is a set of integers representing the nodes that have been visited during the Breadth-First Search. Finally, an array of integer $matched[]$ of size $|U| + |V|$ is used to indicate matched edges, $matched[i] = j$ meaning that the current matching contains the edge (i, j) (of course, we have $matched[j] = i$).

Algorithm 1.2 shows a Breadth-First Search (BFS) within the bipartite graph that is looking for an augmented path from the matching given by array $matched$. Line 3 initialises the path to be empty and Line 4 initialises the queue with the free node i and $last$ as true because we want to start with unmatched edges. While the queue Q is not empty (Line 5), the current searched node $(j, last)$ is removed from Q (Line 6). Given the value of $last$, the set of nodes to visit next in the search are added to set $visit$ (Line 7). If $last$ is true (meaning that last taken edge was matched), we want to visit all neighbours but the matched value, else we need to visit the matched value to guarantee the path is an alternating path. For each node to visit next (Line 12), if this

Algorithm 1.2: Breadth-First Search (BFS) algorithm

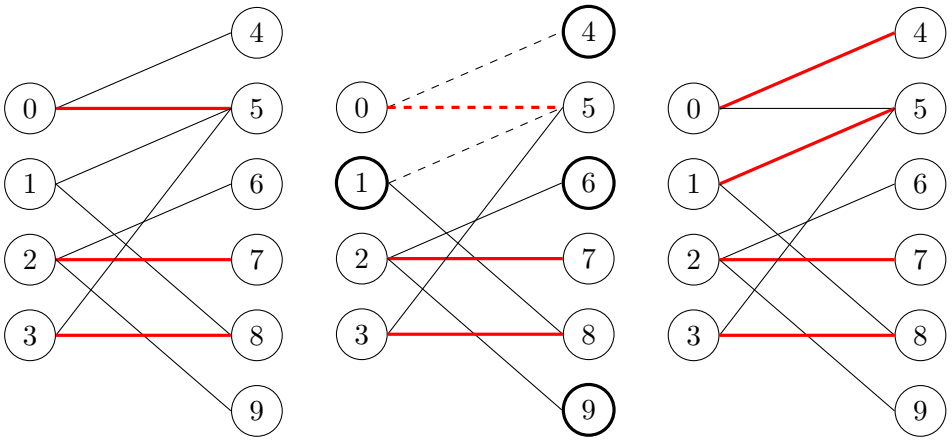
```
1  Function BFS(G = (U, V, E) : bipartite graph, i : integer, F: set of free
     vertices) : integer
2  begin
3      in ← ∅
4      Q.push((i, true))
5      while Q ≠ ∅ do
6          (j, last) ← Q.pop()
7          if last then
8              visit ← N(j) \ {matched[j]}
9          else
10             visit ← {matched[j]}
11         end
12         for k ∈ visit do
13             if k ∉ in then
14                 next[k] ← j
15                 if matched[k] = j then
16                     Q.push((k, true))
17                 else
18                     Q.push((k, false))
19                 end
20                 in ← in ∪ {k}
21                 if k ∈ F then
22                     return k
23                 end
24             end
25         end
26     end
27     return −1
28 end
```

node k is not in the path (Line 13), then we add k to the path (array *next* represents the path in the converse sense it was visited during the search) and add k to Q. If k is a free node (Line 21), the search can be stopped immediately as we have found a new augmenting path. The algorithm returns the value ending the augmenting path, should one was found, and returns -1 otherwise.

Algorithm 1.1 is the main algorithm of the Hopcroft-Karp algorithm. It starts by initiating the matching to be empty (Line 3) and the set of free nodes F as all the nodes (Line 6). *found* is a boolean indicating if an augmenting path was found. So while it is the case (Line 7), we loop and look for a new augmenting path, should one exists. For each free node u in U (Line 9), we call the BFS to see if there is an augmenting path (Line 10). If an augmenting path is found (Line 11), *i.e.* $mate \neq -1$, u and *mate* are removed from the free nodes (Line 13) and the matching is updated (Line 14). *found* is then updated to indicate that an augmenting path was found and the algorithm should loop again (Line 20). Finally, the algorithm returns the size of the maximum matching that was computed (Line 24).

Figure 1.3 shows an example of how to find maximum matching with an augmenting path. The left graph shows the bipartite graph and a matching (represented in red). The middle graph shows free nodes thicker and an augmenting path in dashed

lines. Finally, the right graph shows the resulting matching, which is maximum in this example.



(A) Initial matching (B) Augmenting path (C) Maximum matching

FIGURE 1.3: Example of finding maximum matching with augmenting paths

Micali and Vazirani had later given an algorithm of the same time complexity as the Hopcroft-Karp algorithm that finds a maximal matching in a general graph [MV80].

Chapter 2

Constraint Programming

In order to solve satisfaction and optimisation problems, programming paradigms, such as Linear Programming or Boolean Satisfiability, are used. However, these paradigms come with a certain framework, which could lead to impossibility to express some particular constraints, or hardly doing it with ingenuity. To bypass this difficulty, Constraint Programming (CP) arose in the 1970s and 1980s as a new paradigm to solve satisfaction and optimisation problems, among foundational papers are [Mon74; Lau78; Hen89]. The strength of Constraint Programming is its high expressiveness in the variety of constraints that can be used. In this chapter, we are going to present fundamental notions and notations in Constraint Programming that are necessary to present the contributions described in Part II and Part III. The sections of this chapter add each a key notion of Constraint Programming that, when put together, allow to understand how this framework solves problems. Section 2.1 presents how to model a given problem. Section 2.2 presents very key notions of Constraint Programming, which are the propagation process and the notions of consistencies. Section 2.3 explains how a given model is solved.

Most of the notations we will use in this chapter come directly from the *Handbook of Constraint Programming* [RBW06]. For readers that want to deepen into the notions presented in this chapter, we recommend the reading of this book. Notions in Section 2.1 and Section 2.2 are mostly approached in the chapter by Christian Bessière [Bes06] and the chapter by Christian Schulte and Mats Carlsson [SC06] and the ones in Section 2.3 in the chapter by Peter van Beek [Bee06].

2.1 A flexible modelling paradigm

2.1.1 Basic notions and notations

Constraint Satisfaction Problems (CSP) and *Constraint Optimisation Problems (COP)* are mathematical problems that are described with *variables*, *domains* and *constraints*. The (initial) *domain* $D(x)$ of a variable x is the set of values that can be affected to this variable. Domains are *finite* most of the time. Without loss of generality, finite domains can be mapped to the set $\mathbb{Z}$ of integers, thus we will consider all along this manuscript that we manipulate integer variables whose domain is therefore a finite subset of $\mathbb{Z}$. Constraints specify relations between variables that are allowed. A *solution* of the problem is an affectation of a value to each variable from its domain such that all constraints are satisfied.

Definition 2.1 (Constraint [Bes06])**.** *A* constraint *c is a relation defined on a sequence of k variables $scope(c) = \langle x_1, \ldots, x_k \rangle$. c is the subset of $\mathbb{Z}^{|scope(c)|}$ that contains the combinations of values (or* tuples*) $\tau \in \mathbb{Z}^{|scope(c)|}$ that* satisfy *c. Such a tuple τ is also called a* solution *of the constraint c and we note $\tau \in c$. We note $\tau[x_i]$ the assigned value to x_i in a tuple solution.*

A constraint can be either specified *extensionally* by giving the set of its satisfying tuples, either *intensionally* by a mathematical formula. For instance, we can define the constraint $c_1(x, y, z) \equiv ((1, 1, 2), (1, 2, 2), (2, 3, 3))$ extensionally. The constraint c_1 is satisfied only when the variables x, y and z take simultaneously values that correspond to one of the listed tuples. $x = 1, y = 1, z = 2$ is a solution of the constraint, whereas $x = 1, y = 1, z = 1$ is not a solution of the constraint. The constraint ALLDIFFERENT can be intensionally defined as following: $\text{ALLDIFFERENT}(x_1, \ldots, x_n) \equiv (\bigwedge_{1 \leq i < j \leq n} x_i \neq x_j)$. In other words, no two variables in the scope of the constraint can simultaneously take the same value.

Definition 2.2 (Constraint Network [Bes06])**.** *A* constraint network *is a triplet* $\langle X, D, C \rangle$ *defined as:*

- $X = (x_1, \ldots, x_n)$ *is a finite sequence of integer variables*
- $D = D(x_1) \times \cdots \times D(x_n)$ *is the cartesian's product of the domain of each variable, with* $D(x_i) \subset \mathbb{Z}$ *for all* $x_i \in X$
- $C = \{c_1, \ldots, c_e\}$ *is the set of constraints whose corresponding scopes are subsets of* X*.*

A *Constraint Satisfaction Problem (CSP)* is the problem consisting in looking for a solution of a given constraint network. A *Constraint Optimisation Problem (COP)* is defined as a CSP and an *objective* we want to optimise (either minimise or maximise), that is the minimisation or maximisation of an evaluation function (also called *objective-function*) $f : X \longrightarrow \mathbb{Z}$ of the variables. Any solution whose evaluation value is minimal (or maximal given the objective-function) is an *optimal solution.*

Example 2.1. *Let* x, y *and* z *be three variables whose domains are* $D(x) = \{1, 3, 4\}$, $D(y) = \{2, 3, 4\}$ *and* $D(z) = \{3, 4, 5, 7, 9\}$*. We consider the two constraints* $c_1(x, y) \equiv x \leq y$ *and* $c_2(x, y, z) \equiv 2x + y \leq z$*. This is a CSP, whose solutions are the tuples* $(1, 2, 4), (1, 3, 5), (3, 3, 9)$*. We can build a COP from this CSP by minimising the following objective-function:* $f(x, y, z) = 5x - 2y + z$*. The only optimal solution is* $(1, 3, 5)$ *which evaluates at* 4*. If, for the same CSP, we want to minimise the objective-function* $f(x, y, z) = z$*, then the optimal solution is* $(1, 2, 4)$ *whose evaluation is* 4*.*

Definition 2.3 (Instantiation [Bes06])**.** *Given a constraint network* $N = \langle X, D, C \rangle$*:*

- *An* instantiation I *on* $Y = (x_1, \ldots, x_k) \subseteq X$ *is an assignment of values* $v_1, \ldots, v_k$ *to the variables* $x_1, \ldots, x_k$*, that is* I *is a tuple on* Y*. We will note* $I = \{x_1 \longrightarrow v_1, \ldots, x_k \longrightarrow v_k\}$*.*
- *An instantiation* I *on* Y *is* valid *if for all* $x_i \in Y, I[x_i] \in D(x_i)$*.*
- *An instantiation* I *on* Y *is* locally consistent *iff it is valid and for all* $c \in C$ *with* $scope(c) \subseteq Y$*,* $I[scope(c)]$ *satisfies* c*. If* I *is not locally consistent, it is* locally inconsistent*.*
- *A* solution *to a network* N *is an instantiation* I *on* X *which is locally consistent.*
- *An instantiation* I *on* Y *is* globally consistent *(or* consistent*) if it can be extended to a solution (i.e. there exists a solution to* N *such that* I *is the subset of this solution over* Y*).*

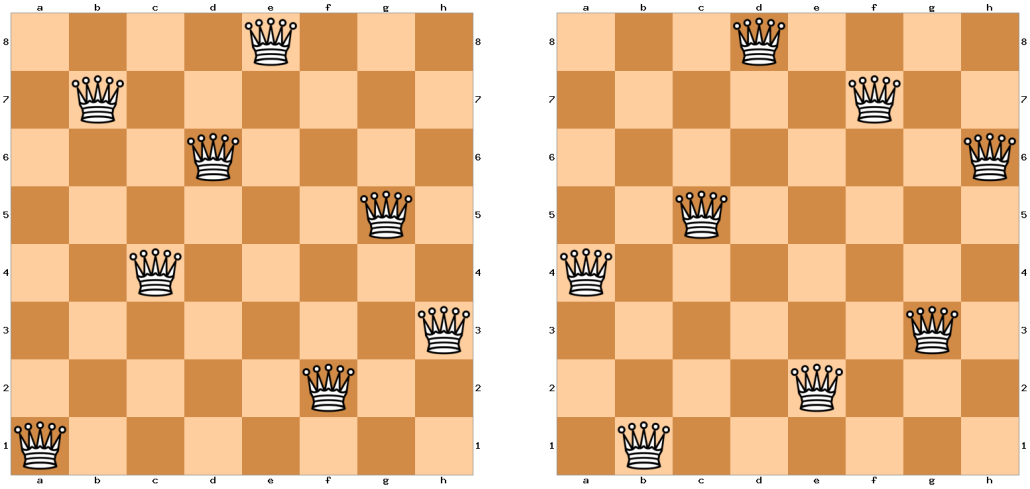
For convenience, and as we will often refer to these notions in this thesis, we introduce here notions and notations important for the rest of the thesis.

Notation 2.1. *The* lower bound *is the smallest value in the domain of the variable and is noted with a line underneath the variable:* $\underline{x} = \min(D(x))$. *The* upper bound *is the greatest value in the domain of the variable and is noted with a line overhead the variable:* $\overline{x} = \max(D(x))$.

A variable x *is* instantiated *whenever its domain contains only one value:* $|D(x)| = 1$.

2.1.2 An example : The N-queens problem

In this section, we show how to model in CP a famous mathematical puzzle: the 8-queens puzzle, and some of its variant. The 8-queens puzzle was first presented by chess composer Max Bezzel in 1848. The idea is to find how to place 8 queens on a chess board so that no two queens threaten each other. As, in chess, the queen threatens all squares within its row, its columns and its diagonals. The puzzle consists in finding how to place the 8 queens on a 8×8 board so that no two pieces share the same row, column or diagonal. Figure 2.1 shows two solutions of the 8-queens puzzle.



(A) Solution 1 (B) Solution 2

FIGURE 2.1: Example of solutions for the 8-queens problem[1]

From this puzzle rapidly arose the more general **N-queens problem**, which consists in finding how to place n queens on a $n \times n$ board so that no two queens threaten each other. In 1969, Hoffman et al. showed that there exist solutions for all natural number n except for 2 and 3 [HLM69]. Let us mention that, in 1991, Chabrier et al. showed how to solve this problem up to 1 million queens [CCT91]. Let us see how to model such a problem.

We introduce variables $b_{i,j}$ for all $(i,j) \in [1,n]^2$ such that $b_{i,j} = 1$ if there is a queen on the square at row i and column j, and $b_{i,j} = 0$ otherwise. Such variables are called *boolean variables*. Figure 2.2 shows one model for the N-queens problem. Equation 2.1 enforces that there can be only one queen on each row. Equation 2.2 is equivalent for the columns. Equations 2.3 and 2.4 enforce that no two queens can be on the same diagonal.

We thus have built a CP model for the general N-queens problem using n^2 variables of domains' size 2 and $4n$ constraints. However, this is not the only model possible to solve this problem. See for instance a model using integer variables in Figure 2.3. In

[1]The author would like to thank Apronus for the authorisation to use their online tool to generate the chess diagrams.

$$\forall i \in [1,n], \sum_{j\in[1,n]} b_{i,j} = 1 \tag{2.1}$$

$$\forall j \in [1,n], \sum_{i\in[1,n]} b_{i,j} = 1 \tag{2.2}$$

$$\forall k \in [1,2n-1], \sum_{\substack{(i,j)\in[1,n]^2 \\ n-j+i=k}} b_{i,j} \leq 1 \tag{2.3}$$

$$\forall k \in [2,2n], \sum_{\substack{(i,j)\in[1,n]^2 \\ i+j=k}} b_{i,j} \leq 1 \tag{2.4}$$

$$\forall (i,j) \in [1,n]^2, D(b_{i,j}) = \{0,1\} \tag{2.5}$$

FIGURE 2.2: MIP Boolean model for the N-queens problem

this model, we use n integer variables $x_1, \ldots, x_n$, one for each row, and whose value indicates the column of the queen at the corresponding row. For instance, if $x_3 = 2$, it means that the queen in row 3 is on the square at column 2. Equation 2.6 thus enforces that no two queens can share the same row or column. Equations 2.7, 2.8 and 2.9 together enforce that two different queens cannot be on the same diagonal. This model uses $3n$ variables of domains' size n and $2n + 3$ constraints.

$$\text{ALLDIFFERENT}(x_1, \ldots, x_n) \tag{2.6}$$

$$\forall i \in [1,n], \begin{cases} y_i = i + x_i \\ z_i = i - x_i \end{cases} \tag{2.7}$$

$$\text{ALLDIFFERENT}(y_1, \ldots, y_n) \tag{2.8}$$

$$\text{ALLDIFFERENT}(z_1, \ldots, z_n) \tag{2.9}$$

$$\forall i \in [1,n], D(x_i) = \{1, \ldots, n\} \tag{2.10}$$

FIGURE 2.3: Integer model for the N-queens problem

This is an example that there does not exist one single CP model for a given problem. Moreover, for the integer model presented in Figure 2.3, we have introduced variables whose only purpose is to help to build the model. We thus distinguish *decision variables* ($x_1, \ldots, x_n$ in our example), whose instantiation is the minimal

way to write a solution of the problem, from *auxiliary variables* ($y_1, \ldots, y_n, z_1, \ldots, z_n$ in our example), whose only purpose is to help building the model.

2.1.3 Finding a solution

To find a solution of the problem described by the CSP, one should explore the *search space*, *i.e.* the set of all tuples, which is the cartesian product of the domains of the decision variables. As explained in [Bee06], there are different possibilities to explore the search space, the three most important ones being the *backtracking search*, the *local search* and *dynamic programming*. In the context of this thesis, we are going to present only the basic principles behind backtracking search, even if some of the contributions could easily be applied in local searches.

The backtracking search in Constraint Programming consists in a *search tree*, a node of which corresponding to a state of the domains of the variables. The idea is to recursively branch on uninstantiated variables and observe the effect on the constraints: whenever it is detected that a constraint cannot be satisfied anymore, the exploration of the branch is stopped and we go back to the parent node. This is called a *backtrack*, during which the domains of the variables are restored to the state they were in before branching. More generally, we qualify of *backtrackable* any data structure that is restored during backtrack.

In Section 2.2, we will look deeper into the process that checks whether a constraint can still be satisfied or not after a branching: the *propagation*. During this phase, we will see several mechanisms that remove values v from $D(x)$ whenever assigning x to v leads the current instantiation I on the variables X to be locally inconsistent. In Section 2.3, we will look deeper into the branching process, also called *search*, and most especially on how we select variables and values to branch on.

2.2 Propagation and consistencies

2.2.1 Generalised Arc Consistency

To find a solution to a constraint network, we need to find an assignment of a value to each variable from its domain such that all constraints are satisfied. To avoid losing time on values that cannot lead to a solution, Constraint Programming highly relies on a principle called *constraint propagation*. The idea is to call each constraint to *filter* values in domains of variables in its scope that it identifies as values that do not appear in any of the constraint's solutions. Such values are said to be *inconsistent*.

In Constraint Programming, there exist several types of *consistencies*, that is level of confidentiality that remaining values in variables' domains will lead to a solution of the problem. The oldest and most well-known one is *(generalised) arc consistency*. It is also known as the *domain consistency* and was introduced in [HSD94].

Definition 2.4 ((Generalised) Arc Consistency ((G)AC) [Bes06])**.** *Given a constraint network $N = \langle X, D, C \rangle$, a constraint $c \in C$, and a variable $x \in scope(c)$,*

- *A value $v \in D(x)$ is* consistent *with c in D iff there exists a valid tuple τ satisfying c such that $v = \tau[x]$. Such a tuple is called a* support *for (x, v) on c.*
- *The domain D is (generalised) arc consistent on c for x iff all the values in $D(x)$ are consistent with c in D.*
- *The network N is (generalised) arc consistent iff D is (generalised) arc consistent for all variables in X on all constraints in C.*

- *The network N is arc consistent iff $\emptyset$ is the only domain tighter than D which is (generalised) arc consistent for all variables on all constraints.*

As most of the time, there is no ambiguity on the different notations, we will often say 'constraint c is GAC' instead of 'D is arc consistent on c for all $x \in scope(c)$'. The same abuse of language is done for arc consistency for a value: we will say 'value v is (arc) consistent' instead of 'value v is consistent with c in D'.

Before presenting other consistencies, let us go back to the constraint propagation process. As we said, constraint propagation consists in calling each constraint to remove invalid values from variables' domain. This is done until no more value can be removed from any variables' domain by any constraint. This particular state, where no more value can be removed, is called the *fixpoint*. In [Mac77], Alan Mackworth proposed AC3, an algorithm that achieves (generalised) arc consistency on a network. Whenever this algorithm stops, either we know for sure that the network is inconsistent, either a fixpoint has been reached.

Algorithm 2.1: Revise function

```
1  Function Revise(x : variable, c : constraint) : boolean
2  begin
3      filt ⟵ false
4      foreach v ∈ D(x) do
5          if ∄τ ∈ c s.t. τ[x] = v then
6              D(x) ← D(x) \ {v}
7              filt ⟵ true
8          end
9      end
10     return filt
11 end
```

Algorithm 2.2: AC3 algorithm

```
1  Function PropagateAC3(N = ⟨X, D, C⟩ : constraint network) : boolean
2  begin
3      Q ⟵ {(x_i, c) | c ∈ C, x_i ∈ scope(c)}
4      while Q ≠ ∅ do
5          (x_i, c) ⟵ Q.pop()
6          if Revise(x_i, c) then
7              if D(x_i) = ∅ then
8                  return false
9              else
10                 Q ⟵ Q ∪ {(x_j, c') | c' ∈ C ∧ c' ≠ c ∧ x_i, x_j ∈ scope(c') ∧ i ≠ j}
11             end
12         end
13     end
14     return true
15 end
```

See Algorithms 2.1 and 2.2 for the AC3 algorithm. Algorithm 2.1 takes as parameters a variable and a constraint in which scope it is in and looks for a support for every value in the domain of such a variable according to the constraint. If none can be

found, the value is removed from the variable's domain. Finally, the algorithm returns `true` if it removed at least one value from the variable's domain. Algorithm 2.2 is the main algorithm, which will call Algorithm 2.1, that aims to reach the fixpoint. If a filtering occurs for variable x in constraint c, the domain $D(x)$ of variable x is checked. If it is empty, we have reached a dead end and the algorithm returns false. Otherwise, we want to revise all the other variables in the constraint's scope. The algorithm continues while there are variables to revise. It returns true in the end, meaning that a fixpoint has been reached without finding any contradiction (emptying a domain).

Whenever a fixpoint is reached, if there are uninstantiated variables, then we will have to test and try the different possibilities. This process in described in more details in Section 2.3.

Note that, as AC3 is non optimal, other algorithms have since been proposed to achieve arc-consistency for constraint propagation: AC4 [MH86], AC6 [BC93], AC2001 [Bes+05], AC7 [BFR95; BR97; BFR99], AC8 [CJ98], AC3.2 and AC3.3 [LBH03]. In [Rég05], Régin presented AC-*, a configurable, generic and adaptive arc-consistency algorithm, that allows to combine several techniques used by other existing arc-consistency algorithms at the same time.

2.2.2 Fastening the propagation process with weaker consistencies

In general, assuming $\mathcal{P} \neq \mathcal{NP}$, enforcing GAC is not tractable. However, it does not mean that it is impossible to filter any inconsistent value in practice. Other consistencies have therefore been introduced to characterise the state of the domains of the variables with respect to the constraints' satisfaction. In the context of this thesis, we will not talk about consistencies that are stronger than GAC. More information on them can be found in [Bes06]. Here, we concentrate on consistencies that are weaker than GAC, *i.e.* fewer values are filtered than for GAC, but generally faster to enforce. While enforcing these consistencies can still take exponential time in general, focus is made on theoretical speed of polynomial algorithms.

Definition 2.5 (Consistencies on bounds [Cho+06; Bes06])**.** *Given a constraint network $N = \langle X, D, C\rangle$, given a constraint c, a* bound support *τ on c is a tuple that satisfies c and such that for all $x \in scope(c), \underline{x} \leq \tau[x] \leq \overline{x}$. A bound support in which each variable is assigned a value in its domain is a support.*

- *A constraint c is* bound(Z) consistent (BC(Z)) *iff for all $x \in scope(c)$, $(x, \underline{x})$ and $(x, \overline{x})$ belong to a bound support on c.*
- *A constraint c is* range consistent (RC) *iff for all $x \in scope(c)$, for all $v \in D(x)$, (x, v) belongs to a bound support on c.*
- *A constraint c is* bound(D) consistent (BC(D)) *iff for all $x \in scope(c)$, $(x, \underline{x})$ and $(x, \overline{x})$ belong to a support on c.*

The network N is bound(Z) / range / bound(D) consistent iff all its constraints are bound(Z) / range / bound(D) consistent.

Interestingly, more focus has been made on the bound(Z) consistency than on the bound(D) consistency, especially because finding support can be $\mathcal{NP}$-hard for some constraints. Thus, in the literature, when people talk about *bounds consistency (BC)*, they refer to the bound(Z) consistency.

Example 2.2. *This example is the one depicted in [Bes06] to show differences of filtering between BC(Z), BC(D), RC and GAC. In this example, the constraint network*

is composed of 6 *variables* $x_1, \ldots, x_6$ *of respective domains* $D(x_1) = D(x_2) = \{1, 2\}$, $D(x_3) = D(x_4) = \{2, 3, 5, 6\}$, $D(x_5) = \{5\}$, $D(x_6) = \{3, 4, 5, 6, 7\}$. *The only constraint is* ALLDIFFERENT$(x_1, \ldots, x_6)$. *The state of the domains after BC(Z), BC(D), RC and GAC are enforced on the* ALLDIFFERENT *constraint can be seen in Figure 2.4.*

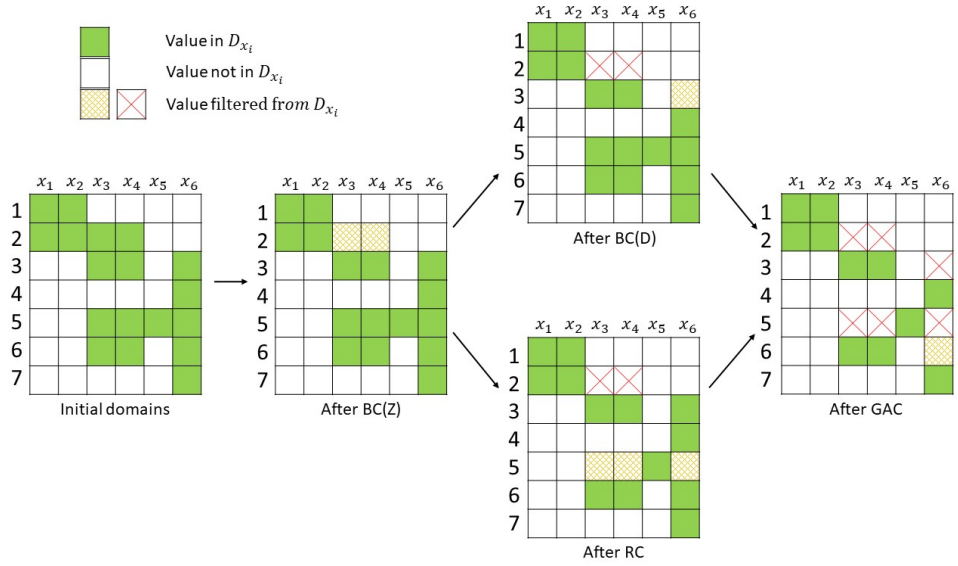


FIGURE 2.4: Propagation difference of BC(Z), BC(D), RC and GAC on the ALLDIFFERENT$(x_1, \ldots, x_6)$ constraint of Example 2.2 [Bes06]

Theorem 2.1 ([Bes06])**.** *Generalised arc consistency is strictly stronger than range and bound(D) consistencies, which are themselves strictly stronger than bound(Z) consistency, which is itself strictly stronger than Forward Checking (see Definition 2.6). Bound(D) consistency and range consistency are incomparable.*

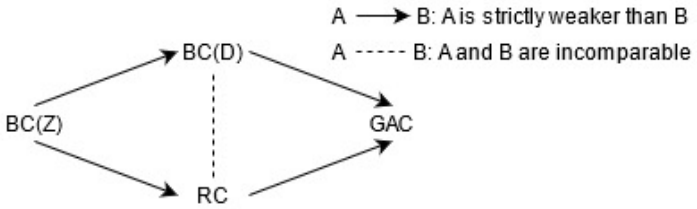


FIGURE 2.5: Graphical representation of the comparison of consistencies' strength

As we have seen, there are several consistencies, and thus different way to apply a constraint's filtering. More specifically, the line 5 of Algorithm 2.1 is modified to respect the wanted consistency's definition. Moreover, specific algorithms have been designed over the years for many different constraints to filter inconsistent values and for different consistencies. These algorithms are a sort of rewriting of Algorithm 2.1 (and most specifically of line 5) and are called *filtering algorithms*.

If we focus on the ALLDIFFERENT constraint for instance, Jean-Charles Régin [Rég94] presented a $O(n^2d^2)$ algorithm that enforces GAC, n being the number of variables in the scope of the constraint and d the number of different values that all variables can take. In 1996, Michel Leconte gave a filtering scheme for the Range Consistency for the ALLDIFFERENT constraint [Lec96]. Later, Mehlhorn and Thiel

[MT00] gave a $O(n \log(n))$ algorithm to enforce Bounds Consistency. Lopez-Ortiz et al. [Lóp+03] also gave a $O(n \log(n))$ algorithm to enforce Bounds Consistency, their algorithm being faster in practice. Furthermore, let us note that in order to allow better solving performance in practice, it might be beneficial to know when to apply such or such consistency to best filter the variables' domains. Boisberranger et al. [Boi+13] studied when it is worthwhile to propagate GAC for the allDifferent constraint instead of only Bounds Consistency.

We would like to remark that it is possible for a filtering algorithm not to have a characterised consistency. For instance, in [Sha04], Paul Shaw gives a filtering algorithm for the BinPacking constraint. His filtering algorithm is better than bound(Z) consistency but does not achieve GAC, neither it does RC. Therefore, it has an undefined consistency, being strictly between BC and RC.

Finally, there is a *trade-off* between the strength of filtering (consistency) and the speed of execution (worst-case time complexity). When solving some problems, it might be interesting to filter less but to explore nodes quicker, either to find at least one solution or even to do the proof of optimality. However, the contrary is also true: stronger filtering can lead to great cuts of the search space and therefore to quickly find a solution or do the proof of optimality. This notion of trade-off is of great importance in practice. For this reason, we will greatly concentrate on this notion all along this thesis.

2.2.3 Fastening the propagation process with fewer calls to the filtering algorithms

Another way to make the propagation process quicker is to avoid calling the filtering algorithms whenever we know that they will have no effect on the domains of the variables. To this end, in [FW91], Freuder and Wallace proposed a technique to estimate when filtering is likely to be effective, therefore reducing the number of calls to constraints' filtering algorithms.

Forward Checking [GB65; HE80] has also been proposed to reduce the number of calls to the Revise function (Algorithm 2.1). The idea is to call the function only on uninstantiated variables x_j that are in the scope of constraints $c \in C$ such that $x_j \in scope(c)$ and such that there is a variable $x_i \in scope(c)$ that has just been instantiated.

Definition 2.6 (Forward Checking [Bes06]). *Let $N = \langle X, D, C \rangle$ be a binary network, i.e. a constraint network with only binary constraints, and $Y \subseteq X$ such that $|D(x_i)| = 1$ for all $x_i \in Y$. N is* forward checking consistent (FC) *according to the instantiation I on Y iff I is locally consistent and for all $x_i \in Y$, for all $x_j \in X \setminus Y$, for all $c(x_i, x_j) \in C$, x_j is arc consistent on $c(x_i, x_j)$.*

FC can be extended to general constraint network in several ways. Van Hentenryck proposed a basic one in [Hen89]: A network is FC according to a partial instantiation I on a subset Y of X iff I is locally consistent and for all $x_j \in X \setminus Y$, for all $c \in C$ such that $scope(c) \setminus Y = \{x_j\}$, x_j is arc consistent on c (formulation from [Bes06]). Bessiere et al. gave five other extension of FC to non-binary constraints [Bes+02].

2.2.4 Global constraints

Earlier, we presented the allDifferent constraint. This constraint is particular as it expresses a certain modelling pattern. For the allDifferent constraint, the

modelling pattern expresses that every two variables in the scope of the constraint must take different values. What is special with such a modelling pattern is that it always has the same definition and authorises variable size of accepted tuples. For instance, allDifferent(x_1, x_2, x_3) or allDifferent$(x_1, x_2, x_3, x_4, x_5)$ are two different constraints, with different size of accepted tuples. However, the inner definition of both constraints is the same: two different variables must take different values. Thus, allDifferent is more a class of constraints rather than a single constraint. Such a class of constraints are usually called *global constraint*. Whenever it is used, a global constraint has a given arity, *i.e.* the size of accepted tuples. Bessiere and Van Hentenryck [BH03] defined several levels of globality for constraints.

Definition 2.7 (Global Constraint [BH03])**.** *A constraint c is* semantically global *if there exists no decomposition of c into constraints of smaller arity.*

A constraint c is operationally global *if there exists no decomposition of c into constraints of smaller arity such that the same level of consistency can be maintained with the decomposition.*

A constraint c is algorithmically global *if there exists no decomposition of c into constraints of smaller arity that can maintain the same level of consistency with the same time and space complexities.*

Global constraints have been heavily developed in the last two decades, authorising a lot of different modelling patterns. All these global constraints allow for an important expressiveness when modelling problems. Beldiceanu et al. made an extensive list of the global constraints that were proposed across the years: the Global Constraint Catalog [Bel+07]. By default, global constraints can be decomposed into simpler constraints, most especially arithmetical constraints, allowing still for basic pruning of inconsistent values. However, to allow for better filtering for global constraints, intensive work is done to propose different dedicated filtering algorithms for global constraints, either to decrease the theoretical worst-case time complexity to enforce a specific consistency, either to propose a filtering algorithm of a given consistency for a given global constraint.

2.2.5 Solver design : improving the propagation

In this section, we focus a bit more on internal mechanics of constraint solvers, especially how to efficiently do constraint propagation. As we have seen, constraints' consistency can widely vary. However, this *heterogeneity* is not supported by the AC3 algorithm and all its improvements and variants. Therefore, the propagation process described in Algorithm 2.1 and Algorithm 2.2 should be revised to authorise such a diversity of constraints' consistency. To this end, we introduce the concept of *propagator*.

Definition 2.8 (Propagator [SC06])**.** *A* propagator *p is a decreasing and monotonic function that maps domains to domains:*

- *p is* decreasing *means that $p(D) \subseteq D$ for all domains D. This property guarantees that constraint propagation only removes values.*
- *p is* monotonic *means that $p(D_1) \subseteq p(D_2)$ whenever $D_1 \subseteq D_2$.*

A propagator p is correct *for a constraint c iff it does not remove any valid assignment for c: $D \cap c = p(D) \cap c$. The* scope *of a propagator is the set of variables it reads and can filter and is noted scope(p) for a propagator p. A propagator p is said to have reached a* fixpoint *whenever $p(D) = D$.*

A constraint can be defined by a set of propagators, each embedding a filtering algorithm. Consistencies of propagators are defined identically as they are for constraints. The set of propagators of a constraint c is noted $prop(c)$. We note P the set of all propagators of a CSP $\langle X, D, C\rangle$: $P = \bigcup_{c \in C} prop(c)$. Algorithm 2.3 shows a basic *propagation engine* [SC06]. This algorithm is robust to the variety of propagators, and most especially to the coexistence of different consistencies at the same time.

Algorithm 2.3: Basic propagation engine

```
1  Function Propagation(N = ⟨X, D, C⟩ : constraint network) : boolean
2  begin
3      Q ⟵ P
4      while Q ≠ ∅ do
5          p ⟵ Q.selectAndRemove()
6          D' ⟵ p(D)
7          if ∃x_i ∈ X s.t. D'_{x_i} = ∅ then
8              return false
9          else
10             M ⟵ {x_i ∈ X | D(x_i) ≠ D'_{x_i}}
11             Q ⟵ Q ∪ {q ∈ P | scope(q) ∩ M ≠ ∅}
12             D ⟵ D'
13         end
14     end
15     return true
16 end
```

Whenever there are propagators to propagate (Line 4), we select a propagator p and remove it from Q (Line 5). The new domain D' is computed (Line 6). If it leads to an empty domain (Line 7), the propagation has failed and the search should backtrack. Otherwise, the set M represents the set of variables whose domain has been filtered by propagator p (Line 10). The set of propagators to propagate Q is updated by adding all propagators whose scope contains variables in M (Line 11). Adding a propagator p to Q is called *scheduling* p. Finally, the domain of the CSP D is updated with the filtered domain after the propagation of p (Line 12). This algorithm returns, as for Algorithm 2.2, true whenever the propagation did not lead to a failure and false otherwise.

Algorithm 2.3 represents a *propagator-oriented* propagation engine because Q contains scheduled propagators. This is not the only way to build a propagation engine: it can also be *variable-oriented*. Initially, Q contains all variables, and whenever a variable is selected and removed from Q all propagators, whose scope it is in, are propagated. All variables whose domain is modified by such propagations are added to Q. The theoretical complexity of both orientations is the same for the propagation engine, however differences of performance can be observed in practice as the size of Q can largely vary depending on the orientation. The difference of order of calling propagators might also lead to differences of performance.

When implementing a constraint solver, optimising the propagation process is an obsession as it is a very frequent step that can be very costly. Indeed, recent solvers can explore thousands of nodes of the search tree every second thanks to their optimised implementations. In the remainder of this section, we are going to browse through some useful techniques to this matter.

Idempotent propagators In order to avoid running high-complexity filtering algorithms multiple times, propagators can be made *idempotent*, which means that running them again will result in no additional filtering. Whenever a propagator p is idempotent, the equation $p(p(D)) = p(D)$ holds for all domains D. Given a filtering algorithm for a constraint c, the easiest way to make a propagator idempotent is to run the algorithm until no more filtering is done. However, it might not be necessary to run it several times. For instance, given the constraint $c(x_1, x_2) \equiv x_1 \leq x_2$ and domains $D(x_1) = \{2, 3, 4, 5\}$ and $D(x_2) = \{1, 2, 3, 4\}$, the propagator whose filtering algorithm is Algorithm 2.4 is already idempotent: running it once is sufficient.

Algorithm 2.4: Idempotent filtering algorithm for the constraint $c(x_1, x_2) \equiv x_1 \leq x_2$

```
1 Function PropagateLeq(x1, x2 : integer variables)
2 begin
3 |   D(x2) ← D(x2) ∩ [x1; ∞[
4 |   D(x1) ← D(x1)∩] − ∞; x2]
5 end
```

Whenever a propagator is idempotent, it should not be scheduled after its own propagation. In Algorithm 2.3, Line 11 should therefore be adapted to avoid to schedule idempotent propagators.

List of priorities Implementing Q as a FIFO (First-In, First-Out) queue is commonly admitted to be a fair treatment of all propagators. However, as said earlier, the order into which the scheduled propagators are called can make great differences of performance. Indeed, it might be smart to run expensive filtering algorithms as late as possible among all scheduled propagators in order to take into account filtering from other propagators. Filtering algorithms with low time complexity can indeed strengthen domains and thus might decrease the complexity of more complex filtering algorithms (as their time complexity might depend, for example, on the size of the domains). Schulte and Stuckey [SS08] proposed to give each propagator a priority. The lower the priority, the earlier the propagator is run. In such a system, multiple fixpoints are reached: all propagators of priorities i and lower should all have reached a fixpoint (and thus be unscheduled) before selecting any scheduled propagator of priority $i + 1$ or more. The number of priorities is up to the solver developers. In [SS08], Schulte and Stuckey use seven priority levels (originating from [LO00]): *unary* $= 0$, *binary* $= 1$, *ternary* $= 2$, *linear* $= 3$, *quadratic* $= 4$, *cubic* $= 5$ and *veryslow* $= 6$. Each priority level's name represents the arity of the propagator and then the asymptotic runtime of the propagator. For example, *binary* priority level is for binary constraints, whereas *quadratic* priority level is for propagators with n variables whose filtering algorithm's asymptotic runtime is $O(n^2)$. They ran experiments showing performance improvements when using that many priority levels, but also showed that good results can also be obtained with greater priority levels (up to 14, two for each original priority, distinguishing in two levels, *low* and *high*) or fewer priority levels (3 priority levels: *unary* $\leftrightarrow$ *ternary*, *linear* $\leftrightarrow$ *quadratic* and *cubic* $\leftrightarrow$ *veryslow*).

Entailment and staged propagators In [SC06], Schulte and Carlsson give several tips to improve the propagation process. Among them, there is the *entailment* of propagators. A propagator is *entailed* whenever it will have no effect on the domain

D: $\forall D' \subseteq D, p(D') = D'$ for an entailed propagator p. For instance, given the constraint $c(x_1, x_2) \equiv 2x_1 \leq x_2$ and the domains $D(x_1) = [1; 3]$ and $D(x_2) = [7; 10]$, it is useless to call an affiliated propagator. Indeed, whatever the value in the domains of x_1 and x_2, it will necessarily respect the constraint. By entailing the corresponding propagator, we avoid running its filtering algorithm again. This property is a stronger one than idempotency in the sense that idempotent propagators are not added to the list of scheduled propagators after their execution whereas entailed propagators are simply never executed again unless backtracking before the point where it became entailed, *i.e.* entailment is a backtrackable property of propagators. Whenever a propagator is not entailed, we said that it is *disentailed*. Entailment and disentailment can be specified for different consistencies.

In [SS08], Schulte and Stuckey even extended this notion to the one of *staged propagators*. Each propagator has a status, a *stage*, which has several purposes: it can indicate if the propagator is entailed or not, but also which filtering algorithm to use if developers do not create a propagator for each filtering algorithm. When scheduled, it is the stage of the propagator that gives its level of priority. Schulte and Stuckey give two main, but not limited to, use cases when staged propagators are better than multiple propagators: costly propagation and constraints for which there are "different propagation methods with different strength and efficiency available" (which is the case of the CUMULATIVE constraint that we will see later).

Event-based propagation Among the first appearances of this technique in the literature is by Laburthe et al. [LO00] for Choco solver. Since then, this technique has proven to be very effective and is therefore implemented in numerous solvers. The idea is to run propagators only where there are useful. For this, modifications of variables are classified and, given a category of modification, a propagator knows if it can indeed filter or not. Variables modifications are generally defined as [LO00; SC06; SS08]: instantiation (inst), increase of lower bound (inc), decrease of upper bound (dec) or removal of a value (rem). Of course, the events overlap, such that at least one change of bounds (inc and/or dec) occurs whenever an instantiation (inst) occurs. Similarly, event rem always occurs whenever one other occurs. From the knowledge of the modifications that occur on each variable, propagators might execute only parts of their filtering algorithm and thus lead to quicker propagation. Propagators only run filtering algorithms that indeed filter some values.

Incrementality *Incrementality* plays an important role in propagators' speed of execution. It is obvious that not having to build from scratch data used in filtering rules can lead to faster propagators. As the data are still needed to determine if filtering should happen or not, updating only parts of these data might sometimes be done and therefore avoid building the data from scratch. For instance, the bipartite graph used for the GAC filtering algorithm for the ALLDIFFERENT constraint [Rég94] does not need to be built from scratch every time the propagator is called. With a backtrackable graph structure as well as an updating algorithm, it is possible to update the bipartite graph quicker for it to be used in the filtering algorithm. Events, described in previous paragraph, play a key role in incrementality as the knowledge they bring helps to quickly update changed parts of data structures. Therefore, several solvers use a loop when executing a propagator: first events are used to update data structures and possibly do basic filtering, and only then the main filtering algorithm is called.

Watched literals In the continuity of event-based propagation and incrementality, *watched literals* have been introduced by Gent et al. [GJM06] to avoid some propagation. Initially, watched literals come from SAT solvers [Mos+01] and consist in having a watcher on uninstantiated variables in a clause and wake up the filtering process only when these variables have been modified. Adapting this technique to Constraint Programming, for each propagator a watcher is added to a list for each pair variable-value. Whenever a variable is modified by other propagators, the filtering algorithm will be called only if certain watchers are not valid anymore, *i.e.* the value is not in the variable's domain anymore. This technique can be very useful, when applicable, to reduce propagators' scheduling and execution.

2.3 Solving process

Once the model is created, it should be solved. The set of all possible tuples for the variables is also called the *search space*. The idea is to explore this search space in order to find a solution, or an optimal solution in the case of a COP. This exploration can be *complete* or *incomplete*: it is complete if it guarantees that a solution will be found if one exists. If a complete search does not find a solution, it means that it does not exist. If a search is not complete, then it is incomplete.

Complete searches offer guarantees but might become less useful when the size of the problem increases. Indeed, in the general case, the bigger the problem (the size of the cartesian product of the domains of the decision variables), the more difficult it might be to solve it. It is most especially true in the context of Constraint Optimisation Problems, where finding solutions can be easy, but finding an optimal one or proving that it is optimal can be very hard in practice. In such particular problems, incomplete searches might be very useful.

2.3.1 Generic search scheme

A backtracking search can be seen as a depth-first exploration of a search tree. Each node of this tree represents a given state of the variables, that is the domains of the variables are currently subset of the initial domains. A branch of the search tree corresponds to the application of a constraint over a variable. The different branches that are created from a node must be exhaustive and mutually exclusive in the case of a complete search. For instance, on the left branch, we might force a variable x_i to be equals to k, whereas on the right branch, we force $x_i \neq k$.

As we said, to each node of the search tree is associated a given state of the domains. *Backtracking* means that when we return back to a node A from a node B, with B being deeper than A in the search tree, the domains of the variables should be restored to the state they were when the search reaches the node A for the first time.

The *naive backtracking algorithm (BT)* [Bee06], also commonly called *Generate and test* search, consists in selecting, at each node, an uninstantiated variable, should one exist, and *branch* on it, that is apply a *branching constraint* (a unary constraint). Then the propagation phase is called. If all decision variables are instantiated, either all the constraints are satisfied and we have found a solution, either at least one constraint is unsatisfied. Else, if the propagation phase did not lead to a dead end, we select another uninstantiated variable and branch on it. If the propagation phase leads to a dead end or if a solution has been found, the search backtracks to a consistent state of the variables.

There are three popular branching strategies that use unary constraints:

- *Enumeration*: there are as many branches as they are values in the domain of the selected variable, each branch being the instantiation of the variable to the corresponding value.
- *Binary choice points*: two branches are created, the first one assigning a value to the selected variable, the second removing the value from its domain.
- *Domain splitting*: the selected variable is not necessarily instantiated here, but its domain is reduced. The mutually exclusive constraints are of the type $x \leq a$ and $x > a$ for a given $a \in D(x)$.

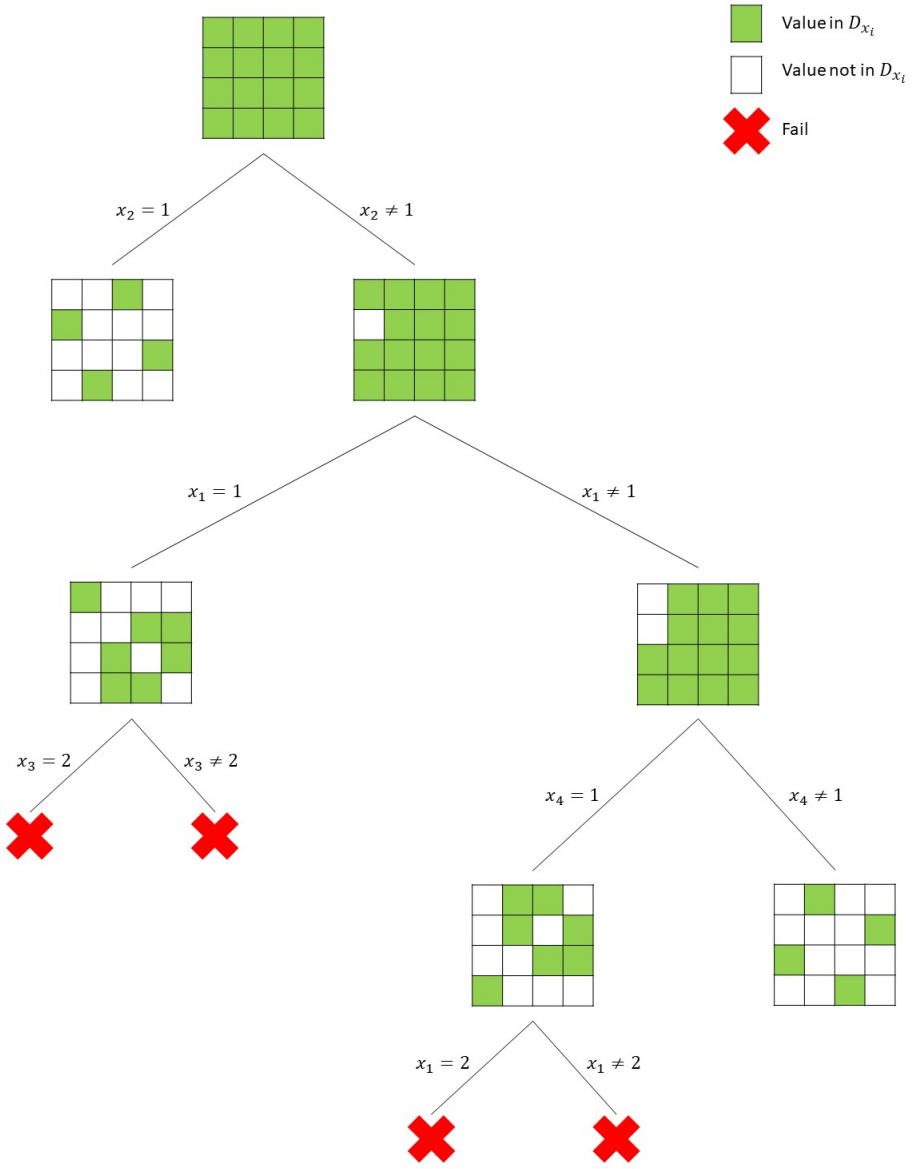


FIGURE 2.6: Example of running a Binary choice points strategy for the 4-queens problem with GAC for the ALLDIFFERENT constraints: each line represents a variable and each column represents a value

Other kind of branching strategies exist that do not post unary constraints, but we are not going to treat them in the context of this thesis. We redirect curious readers

to [Bee06].

The selection of the uninstantiated variable to branch on is done using a *search heuristic* (or simply *heuristic*). This heuristic both selects the variable, as well as the value used in the unary constraint for Binary choice points and Domain Splitting strategies.

2.3.2 Focus on variable- and value-selection heuristics

To improve solving performance, it is better to use a search heuristic that is specific to the problem at hand. However, it might demand a good knowledge and understanding of the problem, which might take time to acquire, when one could want to quickly get first results. To this matter, several generic heuristics have been proposed, with strong results over benchmark with different kinds of problems.

For a long time, the variable-selecting heuristic *dom* [HE80], that consists in selecting the uninstantiated variable with the current smallest domain's size, have long showed good results. The idea behind this heuristic is "To succeed, try first where you are most likely to fail."

Bessiere and Regin [BR96] showed that it can yield better results by selecting the uninstantiated variable with the smallest ratio of the domain's size over the degree of the variable, that is the number of constraints it is involved in.

It has been further improved by Boussemart et al. into the *dom/wdeg* variable selecting heuristic [Bou+04]. At each conflict, the *weight* of each constraint is increased by one. The uninstantiated variable is the one with the smallest ratio of the domain's size over the weighted degree of the variable, the weighted degree of a variable x being defined as the sum of the weight of the constraints that have x in their scope and that has at least one uninstantiated variable in their scope. Intuitively, this search will examine the hard parts of the CSPs first, respecting the first-fail principle.

The *dom/wdeg* variable selecting heuristic has recently been refined to better guide the search [Wat+19]. It is the weighted degree computation that has been refined by increasing it, not only by one, but by a specific value depending on the situation when the conflict arose.

Other kind of generic dynamic heuristics have been proposed and proven to be efficient in practice along the years. For instance, *Impact-based search* [Ref04], *Activity-based search* [MH12] and *Conflict history based search* [HT19] are all based on statistics done on gathered conflicts and value removals, such as for *dom/wdeg*. They differ in the data they gather and how they treat these data. All of them have been proven to be efficient in average and to excel for families of problems.

For instance, Impact-based search have been tuned specifically for scheduling problems [Wol08]. For scheduling problems in particular, Vilim et al. [VLS15] proposed a generic search that can be very efficient, the *Failure-Directed Search (FDS)*, which concentrates on forcing conflicts to close the search tree. Scheduling problems being generally considered as Constraint Optimisation Problems, FDS excels whenever a solution of good quality with respect to the objective has been found. The authors specify themselves that FDS should be used as a "plan B" strategy once the "plan A" strategy is facing difficulties to improve on the best found solution. Also for scheduling problems in particular, Gay et al. proposed the *Conflict Ordering Search (COS)* [Gay+15].

Finally, another generic search heuristic that is worth-noting is *Counting-based search* [PQZ12; GP18; Bia+19] that consists in estimating, at each node of the search tree, the probability that there exists a solution in each possible branch from the node. One can then decide to branch on the lowest or highest probability. Branching on the lowest probability tries to force conflicts in order to cut the search tree quickly, whereas branching on the highest probability aims at finding solutions as quickly as possible.

In the context of this thesis, we do not consider specific techniques such as restarts or backjumping, as these mechanisms can be added to any search strategy and heuristic, and we, once again, redirect the curious reader to [Bee06] as such techniques can be crucial to dynamic searches.

In the remaining of this thesis, we will only consider the Binary choice points strategy, as it is the one implemented by default in Choco solver [PFL17], which will be used for experimentation. A generic scheme for the search process is given in Algorithm 2.5.

Algorithm 2.5: Recursive generic scheme for the search

```
1  Function Search(N = ⟨X, D, C⟩ : constraint network)
2  begin
3      x ← selectVariable(X)
4      if x ≠ null then
5          v ← selectValue(x, D)
6          ⟨X, D′, C′⟩ ← copyCsp(X, D, C)
7          D(x) ← {v}
8          if Propagation(⟨X, D, C⟩) then
9              Search(⟨X, D, C⟩)
10         end
11         D(x)′ ← D(x)′ \ {v}
12         if Propagation(⟨X, D′, C′⟩) then
13             Search(⟨X, D′, C′⟩)
14         end
15     else
16         recordSolution(X, D)
17     end
18 end
```

First, the search procedure starts by selecting an uninstantiated variable x (Line 3). If none can be found, a solution has been found and it is recorded (Line 16). If there is at least one, the selected variable x is selected through a variable-selection heuristic. Then, a value v is selected for x through a value-selection heuristic (Line 5). Once we have selected the variable to branch on and the value on which branching x, a copy of the CSP is done (Line 6). The copy is what makes the backtrack works. As copying is not the only option to allow backtracking, curious readers can find more details of other methods in [Bee06]. The decision is done (Line 7) and is then propagated through the constraint network (Line 8). If a domain has been emptied by the propagation, the search fails and needs to backtrack. Otherwise, the search continues (Line 9). The negated decision represents the other branch and is applied

to the copied constraint network (Line 11). Once again, propagation is done (Line 12) and either leads to a fail or the search can continue for this branch (Line 13).

Chapter 3

Solving scheduling problems with Constraint Programming

3.1 Scheduling problems

There are a lot of different kinds of scheduling problems [Kan76; Pin12; Bru13; SZ15b; SZ15a; Bla+19]. For example, the tasks to be scheduled can be *preemptive* or *non-preemptive* depending on whether they, respectively, can be interrupted or not during their processing. For some scheduling problems, the durations of tasks depend on the resource the task is processed by. This property is sometimes referred to as *multi-modal* problems.

A *scheduling problem* can be generically defined as a problem of deciding how to schedule, if it is possible, a set of tasks on a set of resources. The resources can be of a variety of nature, such as the capacity of machines or workers for instance. Precedence can be enforced between tasks, meaning that the starting time or the ending time must be before or after the starting or ending time of another task or of a set of tasks. The resources can be *renewable* or not, *i.e.* their capacity is fixed or decreases in time depending on the tasks that consume this resource. The objective is generally to minimise a quantity, such as the overall time took to process all the tasks, or the money spent to process all the tasks in a given period, and so on.

3.1.1 Notations for scheduling problems

In this thesis, we will consider *non-preemptive* tasks $\mathcal{T} = \{T_1, \ldots, T_n\}$. Each task T_i has a start time s_i, a duration d_i and an end time e_i. In all the scheduling problems we consider in this thesis, tasks have a *fixed duration* or *processing time*. Thus the treated scheduling problems will consist in determining the *starting time* of each task. The *ending time* of a task is deduced by the relation $s_i + d_i = e_i$.

Each task T_i can be seen as a data structure that aggregates a start time s_i, a duration d_i and an end time e_i. To ease notation and readiness, we will either use the task, its start variable, duration variable, or end variable depending on the use context. Moreover, tasks' ending time and duration will be considered as variables to simplify constraints' readiness: the end variable can be seen as an *offset view* of the start variable in our problems. In Constraint Programming, such a data structure for tasks is often called *task variable*. We will therefore openly use "tasks" or "task variables" for one another as we consider that, for each problem, a task variable is created for each task.

Moreover, we will consider problems with a finite number of renewable resources $R_1, \ldots, R_s$. Each resource R_j has a positive-integer maximal capacity C_j. For each resource R_j, each task T_i has a positive consumption of this resource $h_{i,j}$, that are also called *heights* variables.

Finally, all treated scheduling problems have the same objective, which is to minimise the *makespan* C_{max}, that is the maximum completion time of all the tasks: $C_{max} = \max_{i \in [1,n]} e_i$. The *horizon* Hor is the time-point such that all tasks should be processed between 0 and the horizon. For our problems, the horizon will always be the sum of the processing times of all the tasks: $Hor = \sum_{i=1}^{n} d_i$.

3.1.2 Characterising scheduling problems: Graham notation

Such a wide variety of constraints and objectives defining scheduling problems lead naturally to several hierarchical families of problems that have properties in common. In order to better describe these families of scheduling problems, a three-field notation $\alpha \mid \beta \mid \gamma$ has been introduced by Graham et. al [Gra+79]. In this notation, α specifies the machine environment, β indicates certain job characteristics, and γ denotes the optimality criterion.

In their paper, Graham et al. [Gra+79] consider *jobs*, each of which is composed of a given number of operations, each of which should be processed on a machine. Machines can be identical, in which case every operation can be done on, or not and the processing time of an operation depends on the machine it is processed on. In our case, operations are considered as tasks. The processing time of a job J_j on machine M_i is noted $p_{i,j}$. The total number of machines is noted m.

Graham et al. [Gra+79] decompose α into two values $\alpha = \alpha_1\alpha_2$. They present the different values that α_1 can take:

- $\alpha_1 = \cdot$: single machine ; $p_{1,j} = p_j$
- $\alpha_1 = P$: identical parallel machines ; $\forall i, p_{i,j} = p_j$
- $\alpha_1 = Q$: uniform parallel machines ; $p_{i,j} = q_i p_j$ with q_i the speed factor of machine M_i
- $\alpha_1 = R$: unrelated parallel machines.

In these four cases, jobs can be directly seen as tasks in our vocabulary. Their processing time depends on the problem described by the value of α_1.

- $\alpha_1 = O$: the Open-Shop ; each job is composed of operations $\{o_{1,j}, \ldots, o_{m,j}\}$ for a job J_j, operation $o_{i,j}$ should be executed on machine M_i during $p_{i,j}$ time units. There is no specific order of execution between operations.
- $\alpha_1 = F$: the Flow-Shop ; it is a specialisation of the Open-Shop where the operations form a chain ($o_{1,j}$ is executed before $o_{2,j}$, which is executed before $o_{3,j}$, and so on).
- $\alpha_1 = J$: the Job-Shop ; it is a specialisation of the Flow-Shop where operation $o_{i,j}$ is executed on machine $\mu_{i,j}$, with $\mu_{i-1,j} \neq \mu_{i,j}$.

These three new values of α_1 allow to describe different types of scheduling problems, that the first four could not describe.

Whenever α_2 is given, the number of machines m is finite and equal to α_2. Sometimes, when the number of machines is infinite, an ∞ symbol can be indicated to remove any ambiguity between the case where m is an input of the problem or where m is infinite. Of course, whenever $\alpha_1 = \cdot$, $\alpha_2 = 1$.

When project scheduling came along, a new notation had to be introduced. Project scheduling problems are generalisation of Open-shop problems (and so of Flow-shop and Job-shop problems). In our case, we will use the notation $\alpha = PS$. Curious readers can refer to [SZ15b; SZ15a] for notations for the different cases of project scheduling problems.

As for the α value, β is decomposed into several values, from β_1 to β_6 by Graham et al. [Gra+79] In [SZ15b; SZ15a], β is even decomposed into 13 different values, a value β_i in [Gra+79] meaning possibly something different from the same β_i in [SZ15b; SZ15a]. Here, we will present the decomposition proposed by Graham et al. [Gra+79], and invite curious readers to look at [SZ15b; SZ15a] for a wider vision, especially to describe project scheduling problems. The values for $\beta = \beta_1\beta_2\beta_3\beta_4\beta_5\beta_6$ means the following:

- $\beta_1 \in \{pmtn, \cdot\}$: $\beta_1 = pmtn$ means that tasks are preemptive, that is they can be interrupted in their execution and finished later, whereas $\beta_1 = \cdot$ means that tasks are non-preemptive.

- $\beta_2 \in \{res, res1, \cdot\}$: $\beta_2 = res$ supposes that there are s resources $\{R_1, \ldots, R_s\}$, each resource R_k has a capacity C_k and each task T_j uses an amount $h_{j,k}$ of resource R_k when being processed, the resources are supposed to be renewable ; $\beta_2 = res1$ means that there is only one resource, and $\beta_2 = \cdot$ means that there are no constraint resources.

- $\beta_3 \in \{prec, tree, \cdot\}$: $\beta_3 = prec$ means that there exists a precedence graph between jobs or tasks, meaning that a task should be processed before another one if it is a predecessor of the second task in the precedence graph ; $\beta_3 = tree$ means that the precedence graph is a tree where either each node has an outdegree of at most 1, either each node has an indegree of at most 1 ; $\beta_3 = \cdot$ means that there are no particular precedence constraints.

- $\beta_4 \in \{r_j, \cdot\}$: $\beta_4 = r_j$ means that release dates, *i.e.* a task T_j cannot be processed before its release date, are given for each task, while $\beta_4 = \cdot$ means that each task can be processed from time $t = 0$.

- $\beta_5 \in \{m_j \leq \overline{m}, \cdot\}$: $\beta_5 = m_j \leq \overline{m}$ means that there is a constant upper bound of the number of operations of each job J_j (only when $\alpha_1 = J$), while $\beta_5 = \cdot$ means that no such bound is specified.

- $\beta_6 \in \{p_{i,j} = 1, \underline{p} \leq p_{i,j} \leq \overline{p}, \cdot\}$: $\beta_6 = p_{i,j} = 1$ means that all tasks have a unit processing time ; $\beta_6 = \underline{p} \leq p_{i,j} \leq \overline{p}$ means that constant lower and upper bounds are given for the processing time $p_{i,j}$; $\beta_6 = \cdot$ means that there are no specific constraints on the processing times.

It is common not to precise the values taken by $\beta_1, \ldots, \beta_6$ when the value is $\cdot$. As the possibility for each β_i is listed and there is no redundant notations, problems can be noted without ambiguity when some $\beta_i = \cdot$ are not noted.

Finally, the value γ specifies the objective function to optimise. As all the treated problems in this thesis aim to minimise the makespan, we will not go into much detail for the value γ in the notation and report curious readers to the literature [Gra+79; SZ15b; SZ15a]. Let us note that the maximum lateness L_{max}, that is the maximal difference between the end time of a task and its due date, might also appear to be an interesting use case of some contributions presented in this thesis.

Blazewicz et al. [BLK83] extend the specification of the β field such that $\beta = res\lambda\sigma\delta$ indicates that each job requires δ units of up to λ resources, each of capacity σ. A dot "." instead of one such value denotes that it is part of the input.

These notations are standard in the literature to describe scheduling problems and we will use them to describe each of the treated problems.

3.2 Modelling resources in Constraint Programming: a focus on the DISJUNCTIVE and CUMULATIVE constraints

To describe scheduling problems in Constraint Programming, Aggoun and Beldiceanu introduce the CUMULATIVE constraint [AB93]. This constraint has been specifically tailored to model the fact that a set of tasks should not consume more than a certain amount of a renewable resource. Such a constraint can be used to model, for example, the working capacity of a team. Let C be the maximum availability of the concerned resource. The CUMULATIVE constraint can be defined as following:

$$\begin{array}{c} \text{CUMULATIVE}(\{s_1,\ldots,s_n\},\{d_1,\ldots,d_n\},\{e_1,\ldots,e_n\},\{h_1,\ldots,h_n\},C) \\ \Longleftrightarrow \\ \text{CUMULATIVE}(\{T_1,\ldots,T_n\},\{h_1,\ldots,h_n\},C) \\ \Longleftrightarrow \\ \begin{cases} \forall i \in [1,n], s_i + d_i = e_i \\ \forall t \in [0;Hor], \sum\limits_{\substack{i\in[1,n] \\ s_i \leq t < e_i}} h_i \leq C \end{cases} \end{array}$$

The DISJUNCTIVE constraint is a specialisation of the CUMULATIVE constraint where the capacity is equal to 1. The DISJUNCTIVE is therefore defined as:

$$\begin{array}{c} \text{DISJUNCTIVE}(\{s_1,\ldots,s_n\},\{d_1,\ldots,d_n\},\{e_1,\ldots,e_n\},\{h_1,\ldots,h_n\}) \\ \Longleftrightarrow \\ \text{DISJUNCTIVE}(\{T_1,\ldots,T_n\},\{h_1,\ldots,h_n\}) \\ \Longleftrightarrow \\ \begin{cases} \forall i \in [1,n], s_i + d_i = e_i \\ \forall i,j \in [1,n], h_i = 1 \wedge h_j = 1 \implies s_i \geq e_j \vee e_i \leq s_j \end{cases} \end{array}$$

Sometimes, for the DISJUNCTIVE constraint, whenever $h_1,\ldots,h_n$ variables are all fixed to 1, we might remove them from the notation of the constraint to gain in clarity.

3.3 Example of scheduling problems

3.3.1 Resource-constrained Project Scheduling Problem (RCPSP)

The Resource-Constrained Project Scheduling Problem (RCPSP) was introduced in 1970 by Linus Schrage [Sch70] for the form of the problem that interests us. The problem consists of a set of activities (each of which can be seen as a task, thus our notation) $T_1,\ldots,T_n$ and a number of resources s. Each activity T_i has a given fixed duration d_i and is non-preemptive. Activities are subject to *precedence constraints*, meaning that an activity cannot be processed until all its direct predecessors are

entirely processed. The precedence form a directed acyclic graph. Each activity requires a certain amount of only one given resource. Fixed durations and the use of only one given resource (and a fixed amount of resource is consumed by a task) is referred in the literature as the *single-mode* RCPSP. Each resource R_k is renewable, which means that it is entirely recomposed at the end of an activity needing it, and has a given availability C_k. Finally, the objective is to minimise the makespan. We do not consider any release nor due date for all the tasks, which can therefore be processed anytime during $t = 0$ and the horizon.

The RCPSP is denoted $PS \mid prec \mid C_{max}$ in the Graham notation. It is denoted $PS \mid res..., prec \mid C_{max}$ in Blazewicz's notation, which is simply noted $PS \mid prec \mid C_{max}$. The RCPSP was proven $\mathcal{NP}$-hard by Blazewicz et al. [BLK83] as they prove that a reduction of this problem, $P2 \mid res111, chain, p_i = 1 \mid C_{max}$, is already $\mathcal{NP}$-hard.

Here, we will present the simplest CP model for the RCPSP. However, it is interesting to note that state-of-the-art approaches for the RCPSP have used the flexibility of Constraint Programming's solving process to introduce advanced mechanisms to improve filtering on start variables. For instance, Brucker et al. [Bru+98] used the Roy-Floyd-Warshall algorithm [Roy59; Flo62; War62] on the precedence network. No redundant constraints are added to improve filtering, such as DISJUNCTIVE constraint on a set S of tasks that cannot be scheduled simultaneously as described by Baptiste et al. [BP00].

To model an instance of the RCPSP in Constraint Programming, we introduce one task for each activity. For each activity a_i, s_i is the start variable of the activity and e_i its ending time. The $res(\cdot)$ operator is used to indicate the resource required by an activity, and h_i is the corresponding resource consumption for activity T_i (as activities require one and only one resource). Let Γ^- be the set of precedence, represented as pair of integers (i, j) such that activity T_i precedes activity T_j.

$$minimise\ C_{max} = \max_{i \in [1,n]} e_i \quad s.t.$$

$$\forall (i,j) \in \Gamma^-, e_i \leq s_j \tag{3.1}$$

$$\forall k \in [1,s], \text{CUMULATIVE}(\{T_i \mid res(T_i) = R_k\}, \{h_i \mid res(T_i) = R_k\}, C_k) \tag{3.2}$$

FIGURE 3.1: CP Model for the Resource-Constrained Project Scheduling Problem (RCPSP)

3.3.2 Parallel Machine Scheduling Problem with Additional Unit Resources (PMSPAUR)

The Parallel Machine Scheduling Problem with Additional Unit Resources (PMSPAUR) was introduced by Pralet et. al [Pra+14] as a sub-problem of the observation satellites scheduling process. This sub-problem concentrates on the download of files from one satellite to one ground station. Here, we will consider that the download of one file is a task to be scheduled. When the satellite is inside the download window, the ground station opens several communication channels. These channels

are our machines or resources. Each machine can process only one task at a time. All the machines are identical so each task takes as much time to be processed on all machines. The duration of the tasks depends directly on the file's size and is a data of the problem. Finally, each file is stored on one hard-drive and the satellite embeds several hard-drives. Only one file of a hard-drive can be read at a time. These hard-drives are the additional unit resources.

The PMSPAUR is a sub-problem of $P \mid res.11 \mid C_{max}$ where each task is subject to one additional unit resource [Heb+16]. It is $\mathcal{NP}$-hard as it is a generalisation of $P \mid\mid C_{max}$ which is already $\mathcal{NP}$-hard [GJ78; BLK83].

Example 3.1. *Let us consider the following example:* $d_1 = 7$, $d_2 = 2$, $d_3 = 1$, $d_4 = 4$, $d_5 = 3$, $d_6 = 4$, $d_7 = 2$, $d_8 = 6$, $d_9 = 3$ *and* $d_{10} = 2$. *The additional unit resources are distributed as follows:* $R_1 \equiv \{T_1, T_2, T_3, T_4\}$, $R_2 \equiv \{T_5\}$, $R_3 \equiv \{T_6, T_7\}$ *and* $R_4 \equiv \{T_8, T_9, T_{10}\}$. *The number of machines is* $m = 3$.

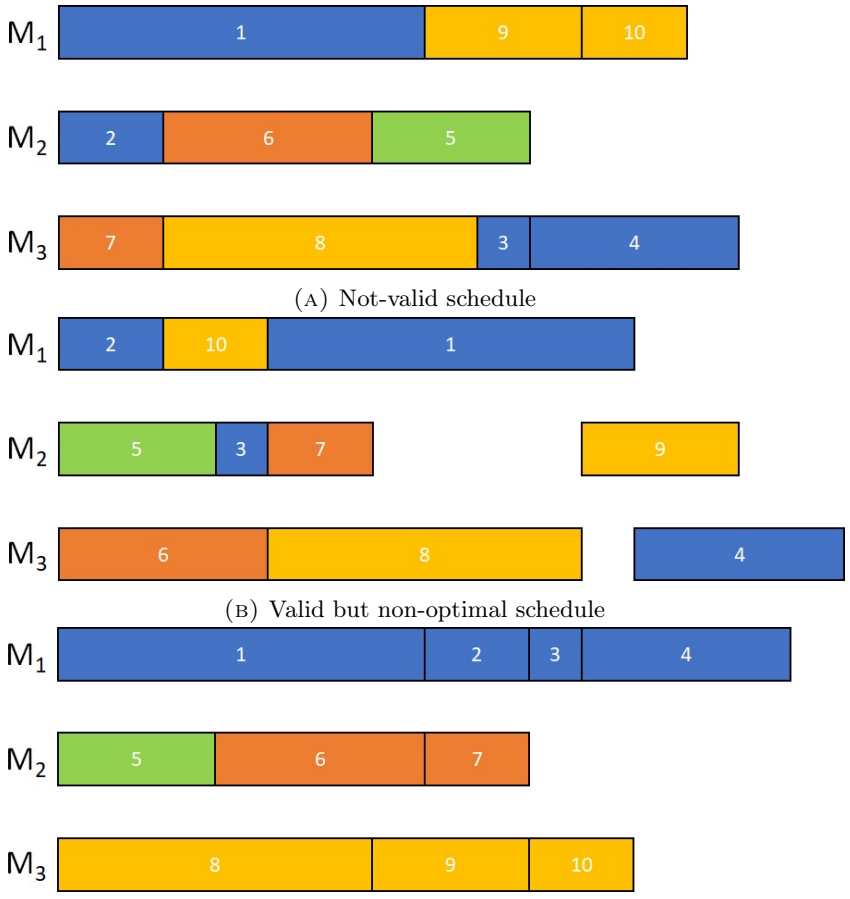
Figure 3.2 shows three examples of schedule for this example. Each task is represented by a rectangle, whose colour depicts the resource (R_1 *in blue,* R_2 *in green,* R_3 *in orange and* R_4 *in gold) and number indicates the id of the task. The schedule in Figure 3.2a is not valid as* T_1 *and* T_2 *are scheduled concurrently while both having* R_1 *as additional unit resource, and* T_8 *and* T_9 *are scheduled concurrently while both having* R_4 *as additional unit resource. The schedule in Figure 3.2b is a valid schedule of makespan* $C_{max} = 15$, *which is not optimal. The schedule in Figure 3.2c shows an optimal schedule of makespan* $C_{max} = 14$.

To the author's best knowledge, the state-of-the-art CP model for the PMSPAUR is given in [God+20] and is presented in the remainder of this subsection. n denotes the number of tasks to process. We note m the number of machines and $R_1, \ldots, R_s$ the additional unit resources. The $res(\cdot)$ operator is used to indicate the additional unit resource of a task. To simplify the notations we write R_j for the set of jobs requiring the resource R_j, *i.e.* $R_j \equiv \{T_i \mid res(T_i) = R_j\} \subseteq \mathcal{T}$. Tasks are necessarily processed, so all the tasks have a height of 1. We take the liberty of not indicating them in the CUMULATIVE and DISJUNCTIVE constraints in the model, which is depicted in Figure 3.3.

3.3.3 Unit Execution Time-Unit Communication Time (UET-UCT)

Let us consider the problem of scheduling tasks subject to precedence on a multi-threaded processor. The idea is to determine when to start processing each task and on which thread. From now on, with respect to common naming in scheduling, we will talk about machines instead of threads. Machines can only process one task at a time.

The problem we described was among the first ones to be researched on in the field of Operations Research, and still have many applications nowadays. This problem has been extended by considering that the result of the processing of a task is unknown from other machines for a certain duration after the end of its processing: it is the time needed to communicate the result to other machines, simply called the *communication time*. If the communication time might be negligible in the case of scheduling tasks on a processor, it is not for other problems. Let us say that you have a very large code database and you want to compile it. In order to save time, you want to parallelise the compilation as much as possible such that the complete process takes as few times as possible (you want to minimise the makespan). You can use a



(A) Not-valid schedule

(B) Valid but non-optimal schedule

(C) Optimal schedule

FIGURE 3.2: Examples of not-valid, valid and optimal schedules for Example 3.1

$$minimise\ C_{max} = \max_{i \in [1,n]} e_i \quad s.t.$$

$$\forall k \in [1, s], \text{DISJUNCTIVE}(\{T_i \mid res(T_i) = R_k\}) \tag{3.3}$$

$$\text{CUMULATIVE}(\mathcal{T}, m) \tag{3.4}$$

FIGURE 3.3: CP Model for the Parallel Machine Scheduling Problem with Additional Unit Resources (PMSPAUR)

set of servers to do it, each one being able to compile one file at a time and automatically communicating the compiled file to other servers. During compilation, some files need to be compiled before to compile some other files (some files have *dependence*), so we still have precedence between tasks. Furthermore, due to the communication times, it might be interesting to compile the same file on several machines: we say that *duplications* are authorised. This problem was the subject of the Final Round of the Google Hash Code competition in 2019[1].

[1] https://storage.googleapis.com/coding-competitions.appspot.com/HC/2019/hashcode2019_final_task.pdf

In this subsection, we will consider a specialisation of this problem where the processing time of each task has a duration of one time unit and the communication times also have a one time unit duration: this is the Unit Execution Time-Unit Communication Time (UET-UCT) problem. We consider that tasks do not have any release nor due date, that the objective is to minimise the makespan C_{max} and that duplication are authorised. In the Graham notation, the considered UET-UCT is noted $P \mid prec, dup, p_i = 1, c_{i,j} = c = 1 \mid C_{max}$, whereas the non-duplication case is noted $P \mid prec, p_i = 1, c_{i,j} = c = 1 \mid C_{max}$. The former one was proven $\mathcal{NP}$-hard by Veltman [Vel93], while the latter one was proven $\mathcal{NP}$-hard by Rayward-Smith [Ray87]. Note that, in the case where the number of machines is at least as high as the number of tasks (we also talk of infinite number of machines), Colin and Chrétienne proved in [CC91] that the problem of minimising the makespan is *polynomial* whenever the maximum communication time is smaller than the the smallest processing time: $\overline{P} \mid prec, dup, c_{i,j} \leq p_i \mid C_{max}$ is in $\mathcal{P}$.

Before giving a CP model for the UET-UCT, we would like to remind a lemma from Munier and Hanen [MH97].

Lemma 3.1 ([MH97])**.** *The subset of schedules for which all copies of a task are performed at the same time is dominating with respect to the makespan.*

Lemma 3.1 helps to reduce the number of variables. Indeed, when authorising duplication, we would have declared one task variable (and subsequent start and end variables) for each task and each machine. This lemma indicates that looking only for schedules when all copies of a task are processed at the same time does not avoid to find an optimal solution of the problem. Therefore, we will use only one task variable for each task T_i and need to decide on which machines it is processed. For this, we need to introduce assignment variables: $b_{i,k} = 1$ whenever task T_i is processed on machine k, and $b_{i,k} = 0$ otherwise. We note m the number of machines. We also consider classic variables in scheduling problems: T_i denotes the task variable for task T_i, whose start variable is noted s_i, end variable is noted e_i and duration is noted d_i. The set of predecessors of a task T_i is noted $\Gamma^-(i)$ and its set of successors is noted $\Gamma^+(i)$. The model is given in Figure 3.4.

Note that Equation 3.6 is the only difference between duplication and no duplication UET-UCT. Here there is an inequality as we authorise duplication, and we would have put an equality if we were not. Equation 3.9 enforces that if a task is scheduled directly after one of its predecessor, this predecessor must be processed on a machine whenever the considered task is (otherwise, the communication will be over and therefore the considered task can be scheduled on any machine). The decision variables of the model are start variables $s_1, \ldots, s_n$ and assignment variables $b_{1,1}, \ldots, b_{n,m}$.

$$minimise\ C_{max} = \max_{i \in [1,n]} e_i \quad s.t.$$

$$\forall i \in [1,n], s_i + 1 = e_i \tag{3.5}$$

$$\forall i \in [1,n], \sum_{k=1}^{m} b_{i,k} \geq 1 \tag{3.6}$$

$$\forall k \in [1,m], \textsc{disjunctive}(\mathcal{T}, \{b_{1,k}, \ldots, b_{n,k}\}) \tag{3.7}$$

$$\forall i \in [1,n], \forall j \in \Gamma^+(i), s_i + 1 \leq s_j \tag{3.8}$$

$$\forall i \in [1,n], \forall j \in \Gamma^+(i), s_i + 1 = s_j \implies \forall k \in [1,m], b_{i,k} \geq b_{j,k} \tag{3.9}$$

FIGURE 3.4: CP Model for the Unit Execution Time-Unit Communication Time (UET-UCT)

Part II

Managing the allDifferent constraint with precedence

Chapter 4

The allDiffPrec constraint and its generalisation

The allDiffPrec is a global constraint defined as an allDifferent constraint and a set of arithmetical constraints $x_i \leq x_j$, also called precedence constraints. It was first introduced by Bessiere et al. in [Bes+11] where they give a bounds(Z)-consistent filtering algorithm, as well as some theoretical results.

In this chapter, we will explore deeper the allDiffPrec constraint. After a more formal definition of the allDiffPrec constraint in Section 4.1 and a presentation of generic filtering schemes for bounds(Z) and range consistencies in Section 4.2, we will review the work of Bessiere et al. in Section 4.3 and Section 4.4. More especially, in Section 4.3, we will discuss on the greedy bound support building algorithm of Bessiere et al. for the different notions it introduces and on which we build further filtering schemes based on Bessiere et al. theoretical results. We will also discuss on the time complexities of these approaches. In Section 4.4, we will concentrate on the BC(Z) filtering algorithm presented by Bessiere et al. in [Bes+11], for which we present faulty behaviours as well as the corrections in the algorithm to fix these behaviours. Based on the same idea underlying the algorithm, we present a similar filtering algorithm.

After this deeper understanding of the state-of-the-art filtering schemes, we introduce a new one in Section 4.5. This new filtering scheme exploits a Lemma in [Bes+11], basically considering domains as sets and not as intervals. We show that this new filtering scheme is strictly stronger than BC(Z) (resp. RC) if it is applied on the bounds of the variables (resp. all the values in the domain). This filtering scheme has a high time complexity, one run of the propagator being in $O(n^3d^2+n^2d^2\sqrt{n+d})$.

Section 4.6 gives an example where none of the filtering schemes are idempotent where domains are sets. Section 4.7 summarises main results on consistencies and time complexities of each filtering scheme. In Section 4.8, we introduce a generalisation of the allDiffPrec constraint, GeneralizedAllDiffPrec, which considers precedence as boolean variables instead of constants. We show that bounds(Z) consistency for the GeneralizedAllDiffPrec can be achieved in $O(n^2)$ amortised time (down a branch of the search tree).

Finally, Section 4.9 summarises main results of the chapter.

The work presented in this chapter was submitted to the CP conference 2021.

4.1 Definition

The allDifferent constraint is probably the most "famous" global constraint. This constraint specifies that all variables in the scope must take different values for the

constraint to be satisfied. This constraint can be used to model lots of different situations. It is not uncommon to see the constraint with precedence constraints on variables in the scope of the allDifferent constraint.

For instance, let us consider the scheduling problem of examination timetabling. Every year, in France, competitive examinations are organised to enter in engineering schools. Generally, every half-day, one and only one exam takes place (it is not entirely true as there are sets of examinations for which two examinations take place the same half-day, but we will not consider such cases to simplify the problem to the case we want to describe). It is common to have two examinations on the same subject, such as mathematics. Between these two exams, one takes place before the other: there is a precedence. To model this problem, we can use a variable for each exam, and the possible values are all the half-days during the week, from 0 (Monday morning) to 9 (Friday afternoon). If variables x_0 and x_1 represent the two mathematical exams, then we have the precedence constraint $x_0 < x_1$.

Other examples are given in [Bes+11], as well as a similar examination timetabling scheduling problem. In [Bes+11], Bessiere et al. introduce the allDiffPrec constraint to represent the modelling pattern consisting of an allDifferent constraint and a set of precedence constraints.

Definition 4.1 (allDiffPrec [Bes+11])**.** *Let $X = \{x_1, \ldots, x_n\}$ be a set of n integer variables, and let $O = \{o_{i,j} \mid 1 \leq i, j \leq n\}$ be n^2 boolean constants representing the precedence. $o_{i,j}$ is* `true` *iff variable x_i precedes variable x_j. As such, $o_{i,i}$ is always* `false`*. Moreover, to avoid cycles in the precedence graph (in which case, there are no solution), we suppose that the formula $\neg(o_{i,j} \wedge o_{j,i})$ is always* `true` *for all $i, j \in [1, n], i \neq j$, and that the precedence relations are transitive: if $o_{i,j}$ and $o_{j,k}$ are* `true`*, then $o_{i,k}$ is* `true`*. The precedence constraints network, represented by variables O, is therefore the transitive closure of the precedence relation. The constraint* allDiffPrec *is defined as:*

$$\text{allDiffPrec}(X, O) \iff \begin{cases} \text{allDifferent}(X) \\ \forall 1 \leq i, j \leq n, o_{i,j} \implies x_i < x_j \end{cases} \tag{4.1}$$

Notation 4.1. *The predecessors of x_j are in the set $\Gamma^-(x_j) = \{x_i \mid o_{i,j}\} \subseteq X$. The successors of x_j are in the set $\Gamma^+(x_j) = \{x_i \mid o_{j,i}\} \subseteq X$. We note Γ^- the set of predecessors sets and Γ^+ the set of successors sets.*

Throughout the chapter, we will discuss about algorithms' worst-case time complexity in terms of two parameters. Of course, the first one is the number of variables n. The second one, noted d, is the number of different values within variables' domains: $d = |\bigcup_{x_i \in scope(c)} D(x_i)|$ if we note c the considered allDiffPrec *constraint.*

4.2 General scheme for BC(Z) and RC filtering

Before talking about the different filtering algorithms for the allDiffPrec constraint, we should see the generic filtering scheme for bounds(Z) and range consistencies, as this scheme can be used with every bound support building algorithms, as it will be the case later in this chapter.

The schemes we are going to present are for a propagator. To define the propagator, we suppose the existence of a function hasBoundSupport(X, x_i, α) that returns

`true` if there exists a bound support for the propagator for the instantiation $x_i \rightarrow \alpha$ and `false` otherwise. This function varies from one propagator to another given the underlying definition of the propagator or of the constraint it represents.

Algorithm 4.1: Generic BC(Z) filtering scheme for a propagator

```
1  Function Propagate(X : variables)
2  begin
3      for x_i ∈ X do
4          while ¬hasBoundSupport(X, x_i, x_i) do
5              D(x_i) ← D(x_i) ∩ [x_i + 1; ∞[
6          end
7          while ¬hasBoundSupport(X, x_i, x̄_i) do
8              D(x_i) ← D(x_i)∩] − ∞; x̄_i − 1]
9          end
10     end
11 end
```

Algorithm 4.2: Generic RC filtering scheme for a propagator

```
1  Function Propagate(X : variables)
2  begin
3      for x_i ∈ X do
4          for d ∈ D(x_i) do
5              if ¬hasBoundSupport(X, x_i, d) then
6                  D(x_i) ← D(x_i) \ {d}
7              end
8          end
9      end
10 end
```

It is interesting to note that both algorithms have a worst-case time complexity of $O(n \times d \times f(n,d))$ with $f(n,d)$ the complexity of a call to function hasBoundSupport. However, Algorithm 4.1 should be faster than Algorithm 4.2 in practice, which is normal as BC(Z) is weaker than RC. Another point to note is that Algorithm 4.2 should be called only when variables' domains can contain holes. Otherwise, in the case of *interval domains*, Algorithm 4.2 will look for bound support for values within the domain that cannot be removed, even if there is no bound support, as holes are not authorised.

4.3 GreedyBC and GreedyRC : Greedy bound support filtering schemes

4.3.1 Description

Building a bound support for the allDiffPrec constraint is not very different from building a bound support for the allDifferent constraint. In [Bes+11], Bessiere et al. explain how to adapt the algorithm for building a bound support for an instantiation $x_i \rightarrow \alpha$ for $\alpha \in D(x_i)$ for the allDiffPrec constraint.

First, the idea is to compute $m = \min_{\substack{x_i \in X \\ |D(x_i)|>1}} \underline{x_i}$ and $M = \max_{\substack{x_i \in X \\ |D(x_i)|>1}} \overline{x_i}$, as we are going to iterate over all values between these two in increasing order. We also note $\mathcal{D}$ the set of values taken by already instantiated variables: $\mathcal{D} = \bigcup_{\substack{x_j \in X \\ |D(x_j)|=1}} D(x_j)$. Let $v \in [m, M]$ be the current value. We add in the set of available variables X', initiated as empty, all the variables whose lower bound is v. If $v \notin \mathcal{D}$, we compute the minimum upper bound of available variables: $u = \min_{x_i \in X'} \overline{x_i}$. X'' is the subset of X' that contains all variables whose upper bound is u. The bound support building algorithm for the allDifferent constraint would break tie arbitrarily here. In [Bes+11], Bessiere et al. say that only variables in X'' that do not have predecessors in X'' should be selected, as otherwise such a predecessor would later be assigned to a greater value than v and would violate the precedence constraint. The selected variable $x_k \in X''$ is assigned the value v and removed from X'. The algorithm continues until all variables are assigned to a value or until $v = M$.

Of course, for this algorithm to work, x_i should be momentarily instantiated to d. Bessiere et al. [Bes+11] also precise that all domains should be *preprocessed*, that is that for each precedence constraint $x_j < x_k$ we have $\underline{x_j} \leq \underline{x_k}$ and $\overline{x_j} \leq \overline{x_k}$. For these two points, they introduce *domains after direct pruning*, that they preprocess before running the bound support building algorithm.

Definition 4.2 (Domains after direct pruning [Bes+11])**.** *Let $x_v \in X$ and $\alpha \in D(x_v)$. The **domains after direct pruning** of assignment $x_v = \alpha$ are denoted $D^{dp}_\alpha(x_1), \ldots, D^{dp}_\alpha(x_n)$ and defined as:*

$$\forall x_u \in X, D^{dp}_\alpha(x_u) = \begin{cases} \{\alpha\} \text{ if } x_u = x_v \\ [\underline{x_u}, \overline{x_u}] \setminus [\alpha, \overline{x_u}] \text{ if } x_u \in \Gamma^-(x_v) \\ [\underline{x_u}, \overline{x_u}] \setminus [\underline{x_u}, \alpha] \text{ if } x_u \in \Gamma^+(x_v) \\ [\underline{x_u}, \overline{x_u}] \setminus \{\alpha\} \text{ if } \alpha \in \{\underline{x_u}, \overline{x_u}\} \\ [\underline{x_u}, \overline{x_u}] \text{ otherwise} \end{cases} \tag{4.2}$$

Within a call to hasBoundSupport, domains after direct pruning are useful to build and to work on during the procedure. As such, at each call, these domains should be built, preprocessed and used for the algorithm to work. The complete procedure for building a bound support is given in Algorithm 4.3.

This algorithm can fail building a bound support when it detects a violation of a *Hall interval* (Line 14). Using the notion of Hall intervals in filtering algorithms for the allDifferent constraint goes back to the works of Leconte [Lec96] and Puget [Pug98]. A Hall interval is an interval $[a, b]$ such that there exists a set of variables $V_{a,b} = \{x \in X \mid [\underline{x}, \overline{x}] \subseteq [a, b]\}$ whose union of interval domains is of the same size as the Hall interval: $|[a, b]| = |V_{a,b}|$. By a pigeonhole argument, for a given interval $[a, b]$, if $V_{a,b}$ contains a greater number of variables that there are values in $[a, b]$, it means that two variables would take the same value, which would make the allDifferent constraint unsatisfiable: such an interval $[a, b]$ is *a violated Hall interval*. Detecting violations of Hall intervals is a key component of filtering for the allDifferent constraint, and therefore for the allDiffPrec constraint.

Checking that all variables are assigned a value at the end of the algorithm is just an additional verification that everything went well, as the *assign* array should not contain -1 if the algorithm was not stopped earlier. Except for the use of domains

Algorithm 4.3: Greedy algorithm to find a bound support for allDiffPrec

1 **Function** hasBoundSupport(X: set of variables, O: set of precedence, x_i: a variable, α: a value in $D(x_i)$) : boolean
2 **begin**
3 Build the domains after direct pruning $D_\alpha^{dp}(x_u)$ for all $x_u \in X$
4 Preprocess all the domains after direct pruning: $\forall o_{j,k} \in O$,
$o_{j,k} \implies \min(D_\alpha^{dp}(x_j)) \leq \min(D_\alpha^{dp}(x_k)) \wedge \max(D_\alpha^{dp}(x_j)) \leq \max(D_\alpha^{dp}(x_k))$
5 $m \leftarrow \min_{\substack{x_j \in X \\ |D(x_j)|>1}} \min(D_\alpha^{dp}(x_j))$
6 $M \leftarrow \max_{\substack{x_j \in X \\ |D(x_j)|>1}} \max(D_\alpha^{dp}(x_j))$
7 $\mathcal{D} \leftarrow \bigcup_{\substack{x_j \in X \\ |D(x_j)|=1}} D(x_j)$
8 $\forall x_j \in X, assign[j] \leftarrow -1$
9 $X' \leftarrow \emptyset$
10 **for** $v \in [m, M]$ *in increasing order* **do**
11 $X' \leftarrow X' \cup \{x_j \in X \mid \min(D_\alpha^{dp}(x_j)) = v\}$
12 **if** $v \notin \mathcal{D}$ **then**
13 $u \leftarrow \min_{x_j \in X'} \max(D_\alpha^{dp}(x_j))$
14 **if** $u < v$ **then**
15 **return** `false`
16 **end**
17 $X'' \leftarrow \{x_j \in X' \mid \max(D_\alpha^{dp}(x_j)) = u\}$
18 Choose $x_k \in X''$ such that $\Gamma^-(x_k) \cap X'' = \emptyset$
19 $X' \leftarrow X' \setminus \{x_k\}$
20 $assign[k] \leftarrow v$
21 **else**
22 Select $k \in [1, n]$ such that $D(x_k) = \{v\}$
23 $assign[k] \leftarrow v$
24 **end**
25 **end**
26 **return** $\bigwedge_{x_j \in X} assign[j] \neq -1$
27 **end**

after direct pruning and Line 18, the algorithm is exactly the same as the bound support building algorithm for the allDifferent constraint.

4.3.2 Complexity and prerequisites

The function hasBoundSupport described in Algorithm 4.3 has a worst-case time complexity of $O(n^2 d)$ (Greedy 1). However, using a Lemma from Bessiere et al. [Bes+11] can lead to smaller time complexities of the filtering scheme.

Lemma 4.1 ([Bes+11]). *Let* allDifferent *and precedence constraints be bounds(Z) consistent over variables X. Let $x_v = \alpha$, $\alpha \in \{\underline{x_v}, \overline{x_v}\}$ be an assignment of a variable*

x_v to its bound and $D^{dp}_{\alpha}(x_1), \ldots, D^{dp}_{\alpha}(x_n)$ be the domains after direct pruning of $x_v \to \alpha$. Then, $x_v \to \alpha$ is bounds(Z) consistent for the allDiffPrec *constraint iff there is a bound support for $x_v \to \alpha$ for* allDifferent$(x_1, \ldots, x_n)$*, where domains of variables X are $D^{dp}_{\alpha}(x_1), \ldots, D^{dp}_{\alpha}(x_n)$.*

From Lemma 4.1, for an instantiation $x_v \to \alpha$ with $\alpha \in D(x_v)$, we see that finding a bound support for the allDifferent constraint on domains after direct pruning is the same as finding a bound support for the allDiffPrec constraint with original domains. As we are interested in bound support here, we can store the domains after direct pruning as intervals. As such, these domains can be stored in integer arrays. As such, building the domains after direct pruning of $x_v \to \alpha$ can be done in $O(n)$, and enforcing precedence constraints on these domains can be done in $O(n^2)$. From these domains, there are two ways to find a bound support: either use the greedy bound support building algorithm (which is Algorithm 4.3 in which Line 18 is just $x_k \in X''$) that runs in $O(nd)$ (Greedy 2), either runs the BC(Z) filtering algorithm for the allDifferent constraint that runs in $O(n)$ [Lóp+03] (Greedy 3).

All implementations of the greedy approach use domains after direct pruning and run therefore in at least $O(n^2)$. As such, knowing if there exists a bound support for an instantiation $x_v \to \alpha$ is done in $O(n^2 d)$ for Greedy 1, in $O(n(n+d))$ for Greedy 2 and in $O(n^2)$ for Greedy 3. It takes at least $O(n^2)$ to test the existence of a bound support for allDiffPrec (instead of the $O(n)$ announced by Bessiere et al. [Bes+11])[1].

Therefore, using dichotomy as explained in [Bes+11], enforcing BC(Z) for allDiffPrec with the greedy approach can be done in $O(n^3 \log(d))$. For RC, no dichotomy can be used and the worst-case time complexity is therefore $O(n^3 d)$.

Using Algorithm 4.1 for BC(Z) or Algorithm 4.2 for RC leads to a worst-case time complexity of $O(n^3 d)$ for a run.

We note GreedyBC the implementation of the allDiffPrec constraint using Algorithm 4.1 and Algorithm 4.3 (which runs in $O(n^3 d^2)$ in total). Similarly, we note GreedyRC the implementation of the allDiffPrec constraint using Algorithm 4.2 and Algorithm 4.3 (which runs in $O(n^3 d^2)$ in total). In Section 5.2, only these implementations were tested on the complete benchmark. Comparison of the three approaches for the greedy filtering scheme (Greedy 1, Greedy 2 and Greedy 3) is discussed in Section 5.1.

4.4 BessiereEtAl : Bessiere et al. filtering scheme

4.4.1 Description

In the same paper, Bessiere et al. [Bes+11] improve on the greedy bound support building algorithm presented in previous section. Given that precedence constraints and the allDifferent constraint have been filtered with bounds(Z) consistency, they give an algorithm that enforces bounds(Z) consistency for the allDiffPrec constraint. The idea is to update directly the upper bound of each variable instead of looking iteratively for a bound support for each bound.

For a variable x_i, the idea is to loop over the different variables and to assign all non-successors to the lowest available value and to adapt the upper bound of the variable x_i according to values that would be taken by successors. For this, the loop is done over variables by increasing upper bounds, and the upper bound of the treated variable x_i is updated at each step x_j by computing the smallest value b such

[1] This correction was discussed and agreed on with one of the author

that there are as many available values in the open interval $[b, \overline{x_j} + 1)$ as there are successors that have been processed since the beginning of the loop: x_i could not be valued greater than $b - 1$ ($x_i \leq b - 1$). This property of b is the invariant of the algorithm. Algorithm 4.4 is the pseudo-code given in [Bes+11].

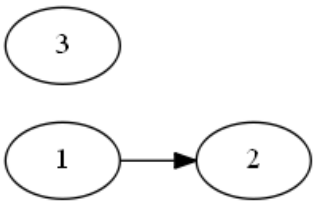
Algorithm 4.4: BC(Z) filtering algorithm for the allDiffPrec constraint from Bessiere et al. [Bes+11]

```
1  Function pruneUpperBounds(X: set of variables, O: set of precedence) :
     void
2  begin
3      Sort the variables by increasing upper bounds into a list L:
       x_{L[idx]} ≤ x_{L[idx+1]}
4      m = min_{x_j ∈ X, |D(x_j)|>1} x_j
5      M = max_{x_j ∈ X, |D(x_j)|>1} x_j
6      for i ∈ [1, n] do
7          Create a disjoint set data structure T with integers from m to M
8          b ← x_{L[1]} + 1
9          for idx ∈ [1, n] do
10             j ← L[idx]
11             if x_j ∉ Γ+(x_i) then
12                 S ← Find(x_j, T)
13                 v ← min(S)
14                 Union(v, max(S) + 1, T)
15             end
16             if idx > 1 then
17                 for k ∈ [1, x_j − x_{L[idx−1]}] do
18                     b ← max(Find(b, T)) + 1
19                 end
20                 if Find(v, T) = Find(b, T) ∧ v > b ∧ x_j ∈ Γ+(x_i) then
21                     b ← min(Find(b − 1, T))
22                 end
23             end
24             D(x_i) ← D(x_i) ∩ ]−∞; b − 1]
25         end
26     end
27 end
```

Bessiere et al. have proven in [Bes+11] that the algorithm described in Algorithm 4.4 enforces bounds(Z) consistency for the allDiffPrec constraint, as long as precedence constraints and the allDifferent constraint have been filtered with bounds(Z) consistency beforehand. Sadly, this pseudo-code has some faulty behaviours.

4.4.2 Fixing faulty behaviours of the algorithm

In this subsection, we will describe some faulty behaviours of Algorithm 4.4 and show how to fix these faulty behaviours. Let us see a first example.



(A) Example 1's precedence graph

Variables	$\underline{x_i}$	$\overline{x_i}$
x_1	0	1
x_2	1	2
x_3	0	2

(B) Variables' domains

FIGURE 4.1: First example where faulty behaviour happens

Example 4.1. *We consider here the example depicted in Figure 4.1. We consider variable $x_i = x_2$. The variables are sorted by increasing upper bound as following: $L = [1, 3, 2]$. We also initialise the union-find T with values $0, 1$ and 2. The domains of the variables are all bounds(Z) consistent. As the first variable in L is x_1 and its upper bound is 1, we initialise b to 2.*

For $j = 1$, x_1 is not a successor of x_2 so we select the set $S \in T$ that contains $\underline{x_1} = 0$ from T: $S = \{0\}$. Thus, $v = \min(S) = 0$. We do the union between values 0 and 1. As it is the first step x_j, we do not execute the rest of the loop. b is still equal to 2, which is the correct value according to the invariant. However, applying the filtering rule would lead x_2's upper bound to be updated to 1, which is incorrect as the domains given in Figure 4.1b are already bounds(Z) consistent.

Remark 4.1. *The body of their pseudo-code is composed of an update of the disjoint data set structure T for non-successors of x_i, an increase of b with a for loop and a decrease of b for successors of x_i as well as in specific cases of update of T.*

First, all the steps but the for loop should be skipped when $x_j = x_i$, as the other steps would make no sense when $x_j = x_i$. Indeed, x_i is not a successor of itself but does not use an available value neither, so no update on T or decrease of b should be applied.

Secondly, x_i should not be filtered until it has been encountered during the loop on variables x_j by increasing upper bound. Indeed, as the variables are sorted by increasing upper bound, no successor of x_i are encountered before x_i as the precedence constraints are supposed to be bounds(Z)-consistent prior to the execution of the algorithm. As such, b is always equal to $\overline{x_j} + 1$ until x_i is encountered, but filtering the upper bound of x_i to at most $b - 1$ at each x_j step would necessarily filter the upper bound of x_i (unless x_i is the variable with the smallest upper bound among all variables in the scope of the constraint).

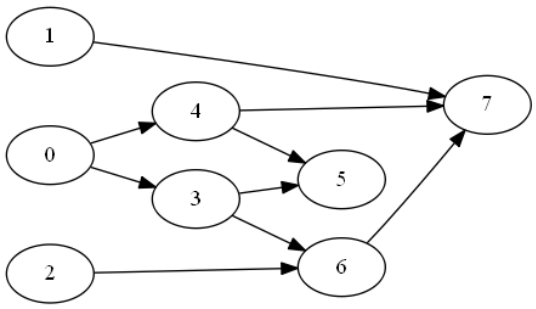
The algorithm's behaviour on Example 4.1 becomes correct with such fixes.

The fixes proposed in Remark 4.1 do not need to be proven as they simply are corrections of non-sense behaviours within the original pseudo-code. Most especially, these fixes make sure that the invariant stays correct at each step, as it should be.

However, it sadly is not enough to completely fix the algorithm. Another faulty behaviour is depicted in the following second example, depicted in Figure 4.2 and Example 4.2.

Example 4.2. *We consider here the example depicted in Figure 4.2. We consider fixes already proposed in Remark 4.1 throughout this example.*

Let us consider the variable $x_i = x_0$. The sorted list of the variables by increasing upper bounds is $L = [0, 3, 2, 1, 4, 6, 7, 5]$. We also initialise the union-find T with values from 0 to 8. b is initialised to $\overline{x_{L[1]}} + 1 = \overline{x_0} + 1 = 3$.



(A) Example 2's precedence graph

Vars	$\underline{x_i}$	$\overline{x_i}$
x_0	0	2 (to filter to 1)
x_1	0	5
x_2	3	4
x_3	1	3
x_4	1	5
x_5	7	8
x_6	4	5
x_7	6	6

(B) Variables' domains

FIGURE 4.2: Second example where faulty behaviour happens

For $j = 0$, following the fixes proposed in Remark 4.1, nothing is done as it is the first step and $x_j = x_i$.

For $j = 3$, x_3 is a successor of x_0, therefore we ignore the first if. As $\overline{x_3} - \overline{x_0} = 1$, b is increased to 4. Finally, since x_3 is a successor of x_0, b is decreased to 3, which is the value it should have at this step according to the invariant.

For $j = 2$, x_2 is not a successor of x_0. We select S that contains $\underline{x_2} = 3$ from T: $S = \{3\}$. Thus, $v = \min(S) = 3$. We do the union between $\{3\}$ and $\{4\}$. Then, as $\overline{x_2} - \overline{x_3} = 1$, b is increased to 5. The last if is ignored as all conditions are false.

However, according to the invariant, b should be equal to 3 at this step of the procedure.

Remark 4.2. *To fix the faulty behaviour described in Example 4.2, b should be updated with the for loop before updating T whenever x_j is not a successor of x_i, otherwise the update of b might not be good with respect to the invariant.*

The algorithm's behaviour on Example 4.2 becomes correct with this fix.

This second fix, described in Remark 4.2, is already covered by the proof given by Bessiere et al. in [Bes+11]: this fix changes the order into which the operations are done such that the updates on b are done on a consistent data structure for the invariant on b to be correct.

All in all, a fixed version of the algorithm of Bessiere et al. is given in Algorithm 4.5. The implementation, noted BessiereEtAl, of the allDiffPrec constraint uses an allDifferent constraint, precedence constraints as well as Algorithm 4.5 to enforce bounds(Z) consistency. Later, we will describe a similar algorithm to update upper bounds of variables based on the same ideas: this later algorithm is not the one used in the implementation noted BessiereEtAl.

4.4.3 Complexity and prerequisites

The worst-case time complexity of Algorithm 4.5 depends greatly on the time complexities of operations Union() and Find(), as well as the time complexity of finding the minimum and maximum values of sets within the disjoint data set structure T. Union() and Find() operations can be done in $O(1)$ [GT85]. Similar implementation techniques can be used to access the minimum and maximum values of each set in T in $O(1)$.

Considering that these four operations on T can each be done in $O(1)$, the fixed version of the algorithm presented by Bessiere et al. (inner behaviour of Algorithm 4.5) has a worst-case time complexity of $O(n + d)$: it is not in $O(nd)$ as the for loop

Algorithm 4.5: Fixed version of the BC(Z) filtering algorithm for the allDiffPrec constraint from Bessiere et al. [Bes+11]

```
1  Function pruneUpperBounds(X: set of variables, O: set of precedence) :
     void
2  begin
3      m ← min_{x_j ∈ X, |D(x_j)|>1} x_j
4      M ← max_{x_j ∈ X, |D(x_j)|>1} x_j
5      for i ∈ [1, n] do
6          Sort the variables by increasing upper bounds into a list L:
             x_{L[idx]} ≤ x_{L[idx+1]}
7          Create a disjoint data set structure T with integers from m to M
8          b ← x_{L[1]} + 1
9          encountered ← false
10         for idx ∈ [1, n] do
11             j ← L[idx]
12             if idx > 1 then
13                 for k ∈ [1, x_j − x_{L[idx−1]}] do
14                     b ← max(Find(b, T)) + 1
15                 end
16             end
17             if j ≠ i then
18                 if x_j ∉ Γ+(x_i) then
19                     S ← Find(x_j, T)
20                     v ← min(S)
21                     Union(v, max(S) + 1, T)
22                 end
23                 if idx > 1 then
24                     if Find(v, T) = Find(b, T) ∧ v > b ∧ x_j ∈ Γ+(x_i) then
25                         b ← min(Find(b − 1, T))
26                     end
27                 end
28             else
29                 encountered ← true
30             end
31             if encountered then
32                 D(x_i) ← D(x_i) ∩ ]−∞; b − 1]
33             end
34         end
35     end
36 end
```

increasing b is done d times over the complete for loop over $idx \in [1, n]$. As it is applied for all variable $x_i \in X$, the worst-case time complexity is $O(n(n + d))$. Bessiere et al. explained how to modify the algorithm such that the time complexity is in $O(n^2)$ [Bes+11].

We remind that Bessiere et al. specify that allDifferent and precedence constraints should have been filtered with bounds(Z) consistency before calling their algorithm in order to enforce bounds(Z) consistency for the allDiffPrec constraint [Bes+11].

4.4.4 Similar algorithm to prune upper bounds

Following on the ideas presented by Bessiere et al., a similar algorithm can be written. In this algorithm, the invariant is almost the same as the one of the original algorithm. This similar algorithm is given in Algorithm 4.6.

Algorithm 4.6: Other algorithm from the idea of Bessiere et al. [Bes+11]

```
1  Function PruneUpperBounds(X: set of variables, O: set of precedence) :
     void
2  begin
3      for i ∈ [1, n] do
4          Sort variables in a list L such that x̄_{L[k]} ≤ x̄_{L[k+1]}
5          Create a disjoint set data structure T with the integers
             [min(⋃_{x_j∈X} D(x_j)), max(⋃_{x_j∈X} D(x_j))]
6          nbSucc ← 0
7          for k ∈ [1, n] do
8              j ← L[k]
9              if i ≠ j then
10                 if j ∉ Γ⁺(x_i) then
11                     S ← Find(x_j, T)
12                     Union(min(S), max(S) + 1, T)
13                 else
14                     nbSucc ← nbSucc + 1
15                 end
16             end
17             if nbSucc > 0 then
18                 a ← x̄_j
19                 tmp ← 1
20                 while tmp ≤ nbSucc ∧ a ≥ x_i do
21                     S ← Find(a, T)
22                     if S = ∅ then
23                         a ← x_i − 1
24                     else
25                         a ← min(S) − 1
26                     end
27                 end
28                 D(x_i) ← D(x_i)∩]−∞; a]
29             end
30         end
31     end
32 end
```

Let us describe the behaviour of this other filtering algorithm (Algorithm 4.6). The variables can be filtered independently, as in the original algorithm. Let us

consider that we are filtering variable x_i. The idea, as in the original algorithm, is to loop over the variables by increasing upper bounds and to look if there are enough available values for the successors of x_i, or to prune x_i's upper bound otherwise. To this matter, when looping on the k^{th} variable in the list $x_j = x_{L[k]}$, either x_j is a successor of x_i and the number of encountered successors is incremented by 1, either x_j is not a successor of x_i and we attribute it the smallest available value from $\underline{x_j}$. Attributing available values is done, as in the original algorithm, with Find() and Union() operations. After this, we compute a the greatest available value for x_i. Within T, available values for successors of x_i are only values that are the greatest value of a set. To compute the highest available value for x_i, we decrease a, as many times as there are encountered successors from the beginning of the loop treating x_i, to the next available value, that is to $\min(Find(a, T)) - 1$ (or to $\underline{x_i} - 1$ if a is not in T). The test on Line 22 is here mostly to assure the good behaviour of the algorithm, as $Find(a, T)$ will return $\emptyset$ if $a \notin T$.

This algorithm has a worst-case time complexity of $O(n^2)$ for a single variable $x_i \in X$, and therefore a worst-case time complexity of $O(n^3)$ in total. Algorithm 4.5 is therefore quicker as the invariant b is updated incrementally.

4.4.5 Discussion on filtering lower bounds

In contrast with Algorithm 4.3, Algorithm 4.5 and Algorithm 4.6 filter only upper bounds of variables. To filter lower bounds of variables, mirror filtering can be applied. For this, either a mirror version should be implemented in order to filter lower bounds (which leads to more development work to implement, test and maintain this other piece of code), either we can consider negative version of variables $y_i = -x_i$ and as such filtering the upper bound of y_i is equivalent to filter the lower bound of x_i. Both versions are good possibilities for implementations within a constraint solver. In our case, we consider the second version: integer arrays are maintained with lower and upper bounds of variables (either the true variables $x_1, \ldots, x_n$ or their mirror versions $y_1, \ldots, y_n$) before applying the filtering algorithm. When filtering lower bounds, the precedence constraints are also inverted: y_i precedes y_j if and only if x_j precedes x_i.

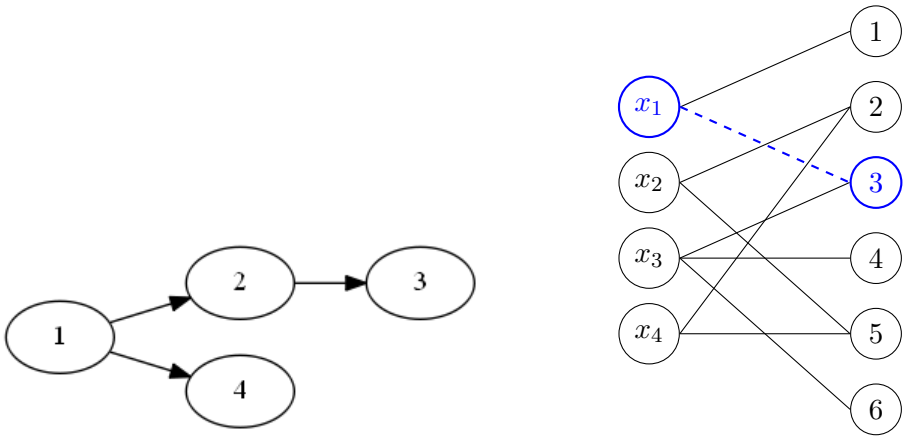
4.5 GodetBC and GodetRC : Better filtering when holes are authorised in domains

4.5.1 Arc-inconsistent but range-consistent values for the allDiffPrec constraint

Remember that Bessiere et al. prove that enforcing domain consistency for the allDiffPrec constraint is $\mathcal{NP}$-hard [Bes+11]. As such, the algorithms presented in previous sections do not consider potential holes within domains. Indeed, as all the algorithms relied on building bound support, only the bounds of the domains are read, despite GreedyRC can create holes in domains.

However, it is possible, in some cases, considering holes in domains, to detect that some bound supports could not be extended to supports.

Example 4.3. *The Figure 4.3 presents an example where a range-consistent value can be safely removed as it is not part of any support. The Figure 4.3a shows the precedence graph and the Figure 4.3b shows the variables' domains in the form of a value graph. The domains are consistent with the precedence constraints and the*



(A) Precedence graph (B) Variables' domains

FIGURE 4.3: Situation where a range-consistent value can be removed.

allDifferent *constraint with bounds(Z) consistency (they even are with generalized arc-consistency in this example).*

The value 3 *should be removed from* $D(x_1)$*, which is why the edge is a dotted line, as it is not part of any support. However, Range-consistency will not remove the value because there exists a* bound support *for it:* $\{x_1 \to 3, x_2 \to 4, x_3 \to 6, x_4 \to 5\}$.

4.5.2 Improving the filtering strength for the allDiffPrec constraint

We will now see how to detect such cases in order to remove such range-consistent values that are not generalized-arc consistent.

Notation 4.2. *From here, we will note the notation of domains after direct pruning to a more suitable form, that could be extended to several instantiations:*

$$\forall x_u \in X, D_{x_v \to \alpha}(x_u) = \begin{cases} \{\alpha\} \text{ if } x_u = x_v \\ D(x_u) \setminus [\alpha, \overline{x_u}] \text{ if } x_u \in \Gamma^-(x_v) \\ D(x_u) \setminus [\underline{x_u}, \alpha] \text{ if } x_u \in \Gamma^+(x_v) \\ D(x_u) \setminus \{\alpha\} \text{ otherwise} \end{cases} \tag{4.3}$$

Domains after direct pruning of $x_v \to \alpha$ *are noted* $D_{x_v \to \alpha}$ *and they are supposed to be preprocessed:*

$$\forall o_{j,k} \in O, o_{j,k} \implies \begin{cases} \min(D_{x_v \to \alpha}(x_j)) < \min(D_{x_v \to \alpha}(x_k)) \\ \max(D_{x_v \to \alpha}(x_j)) < \max(D_{x_v \to \alpha}(x_k)) \end{cases} \tag{4.4}$$

This new notation considers holes in domains. Domains after direct pruning of instantiations $x_i \to v_i$ *and* $x_j \to v_j$ *will be noted* $D_{x_i \to v_i, x_j \to v_j}$.

Definition 4.3 (bipartite graph after direct pruning)**.** *Let* $x_v \in X$ *be a variable and* $\alpha \in D(x_v)$ *be a value in the domain of* x_v*. We define the bipartite graph induced by* $x_v \to \alpha$ *by :*

$$G_{x_v \to \alpha} = (U, V, E) \text{ such that } \begin{cases} U = X \\ V = \bigcup_{x_w \in X} D(x_w) \\ E = \{(x_u, k) \mid x_u \in U, k \in D_{x_v \to \alpha}(x_u)\} \end{cases} \tag{4.5}$$

It is interesting to note that $G_{x_v \to \alpha}$ is a subgraph of the *value graph* introduced by Regin [Rég94].

We note $M(\cdot)$ the operator that gives a maximum matching of a graph given in parameter.

Lemma 4.2. *Let $x_v \in X$ and $\alpha \in D(x_v)$. A maximal matching $M(G_{x_v \to \alpha})$ of size n means that there exists a bound support for $x_v \to \alpha$ for the* allDiffPrec *constraint.*

Proof. If $M(G_{x_v \to \alpha})$ is of size n, then $M(G_{x_v \to \alpha})$ is a support for the allDifferent constraint where domains of variables $x_1, \ldots, x_n$ are $D_{x_v \to \alpha}(x_1), \ldots, D_{x_v \to \alpha}(x_n)$. As $\forall x_u \in X, D_{x_v \to \alpha}(x_u) \subseteq D_\alpha^{dp}(x_u)$, $M(G_{x_v \to \alpha})$ is therefore a support for the allDifferent constraint where domains of variables $x_1, \ldots, x_n$ are $D_\alpha^{dp}(x_1), \ldots, D_\alpha^{dp}(x_n)$. As a support is by definition a bound support, we thus have a bound support for $x_v \to \alpha$ for allDifferent$(x_1, \ldots, x_n)$, where domains of variables X are $D_\alpha^{dp}(x_1), \ldots, D_\alpha^{dp}(x_n)$. By Lemma 4.1, there exists a bound support for $x_v \to \alpha$ for the allDiffPrec constraint. □

Lemma 4.3. *Given that precedence constraints are bounds(Z) consistent, a bound support of* allDifferent(X) *can be transformed into a bound support for* allDiffPrec(X, O).

Proof. Let us consider a bound support for allDifferent(X). Either all precedence constraints are satisfied and the bound support for allDifferent(X) is also a bound support for allDiffPrec(X, O), either at least one precedence constraint is violated within the bound support.

Let us consider two variables x_i and x_j such that we have the precedence $x_i < x_j$. We note β (respectively γ) the value assigned to x_i (respectively x_j) in the bound support for allDifferent(X). Let us suppose that $\beta > \gamma$, such that the constraint is violated (as we are studying how to transform the bound support for allDifferent(X) into a bound support allDiffPrec(X, O)). Since the precedence constraints are supposed to be consistent, then we have $\underline{x_i} < \underline{x_j}$ and $\overline{x_i} < \overline{x_j}$. Since β is assigned to x_i, then $\underline{x_i} \leq \beta \leq \overline{x_i}$. Similarly, $\underline{x_j} \leq \gamma \leq \overline{x_j}$. Finally, since $\beta > \gamma$, we have $\underline{x_i} < \underline{x_j} \leq \gamma < \beta \leq \overline{x_i} < \overline{x_j}$. Therefore, $\beta \in [\underline{x_j}, \overline{x_j}]$ and $\gamma \in [\underline{x_i}, \overline{x_j}]$. As such, we could assign β to x_j and γ to x_i.

For each precedence constraint violated by the assignment built from the bound support for allDifferent(X) or that becomes violated from a swap, we swap the values of the two variables in the assignment. When there are no more violated constraints, the built assignment is therefore a bound support for allDiffPrec(X, O): the allDifferent constraint is satisfied as we have n different values (as we started from a bound support for allDifferent(X)) and all precedence constraints are satisfied after swapping when needed. □

We propose the following proposition as a filtering rule.

Proposition 4.1. *Let $x_v \in X$ and $\alpha \in D(x_v)$. If there is no maximum matching of size n in $G_{x_v \to \alpha}$, then α is inconsistent with* allDiffPrec *in D. The filtering rule can be expressed as:*

$$|M(G_{x_v \to \alpha})| < n \implies D(x_v) \leftarrow D(x_v) \setminus \{\alpha\} \tag{4.6}$$

Proof. From Lemma 4.2, the existence of a maximum matching $M(G_{x_v \to \alpha})$ of size n implies the existence of a bound support for $x_v \to \alpha$.

By contraposition, suppose that $\alpha \in D(x_v)$ is arc-consistent with allDiffPrec in D. As such, for all $i \in [1, n]$, there exists $d_i \in D_{x_v \to \alpha}(x_i)$ such that all these values satisfy the precedence constraints and $d_i \neq d_j$ for all $i \neq j$. As $d_i \in D_{x_v \to \alpha}(x_i)$ for all $i \in [1, n]$, these assignments correspond to a maximum matching $M(G_{x_v \to \alpha})$ of size n. □

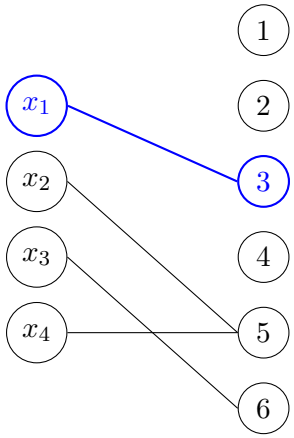


Figure 4.4: Representation of $G_{x_1 \to 3}$ corresponding of example depicted in Figure 4.3

Example 4.4. *Let us see the behaviour of Proposition 4.1 on the Example 4.3.*

Let us see how to build the bipartite graph $G_{x_1 \to 3}$. First, all edges corresponding to other values than 3 should be removed from the value graph for variable x_1: $(x_1, 1)$. The edges from any other variable than x_1 to the value 3 are also removed: $(x_3, 3)$. Finally, all precedence should be enforced within the graph. As x_1 precedes x_2, the edge $(x_2, 2)$ is thus removed, which in turn removes edge $(x_3, 4)$ as x_2 precedes x_3. Similarly, the edge $(x_4, 2)$ is removed as x_1 precedes x_4. All precedence are now respected within the graph. The result is shown in Figure 4.4.

However, in $G_{x_1 \to 3}$, there is no matching of size 4. That means that, from Proposition 4.1, there exists no support for the assignment $x_1 \to 3$. Therefore, we know that we can remove 3 from $D(x_1)$.

Theorem 4.1. *Applying the filtering rule from Proposition 4.1 on all variables $x_v \in X$ and value $\alpha \in \{\underline{x_v}, \overline{x_v}\}$ achieves bounds(Z) consistency on the* allDiffPrec *constraint.*

Proof. The proof is immediate with Lemma 4.2 and Proposition 4.1. □

Theorem 4.2. *Applying the filtering rule from Proposition 4.1 on any variable $x_v \in X$ and value $\alpha \in D(x_v)$ achieves range consistency on the* allDiffPrec *constraint.*

Proof. The proof is immediate with Lemma 4.2 and Proposition 4.1. □

4.5.3 Complexity and prerequisites

Theorem 4.3. *Computing the existence of a bound support for $x_v \to \alpha$ for the* allDiffPrec *constraint using the filtering of Proposition 4.1 can be done in $O(n^2 d + nd\sqrt{n+d})$ time with $d = |\bigcup_{x_v \in X} D(x_v)|$.*

Proof. Building the domains after direct pruning can be done in $O(nd)$, while their preprocessing can be done in $O(n^2 d)$.

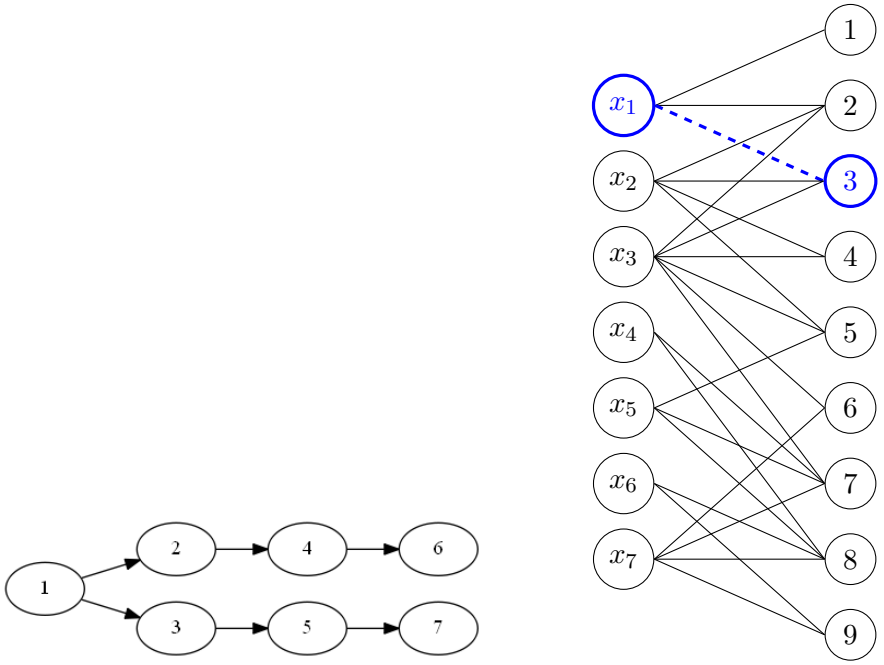
A maximum matching of a bipartite graph of k nodes and m edges can be computed in $O(m\sqrt{k})$ by using the Hopcroft-Karp algorithm [HK73]. It can therefore be done, for a given variable $x_v \in X$ and $\alpha \in D(x_v)$, in $O(nd\sqrt{n+d})$ for each bipartite graph $G_{x_v \to \alpha}$. From Lemma 4.2, we know that if this maximum matching is of size n, then there exists a bound support for $x_v \to \alpha$ for the allDiffPrec constraint. □

It is interesting to note that it is not necessary to previously filter the allDifferent constraint for our approach to work. Indeed, a maximal matching $M(G_{x_v \to \alpha})$ of size n is a support for $x_v \to \alpha$ for the allDifferent constraint.

We note GodetBC the implementation of the allDiffPrec constraint using precedence constraints as well as Algorithm 4.1 and Proposition 4.1 as the hasBoundSupport function. Similarly, GodetRC represents the implementation of the allDiffPrec constraint using precedence constraints as well as Algorithm 4.2 and Proposition 4.1 as the hasBoundSupport function.

4.5.4 Weaker filtering than GAC

As we said at the beginning of the previous subsection, Bessiere et al. show that enforcing GAC for the allDiffPrec constraint is $\mathcal{NP}$-hard [Bes+11]. We will give here an example of an arc-inconsistent value that is not removed by our approach. This example is depicted in Figure 4.5 and consistency is discussed in Example 4.5.



(A) Precedence graph (B) Variables' domains

Figure 4.5: Situation where the filtering rule of Proposition 4.1 does not remove an arc-inconsistent value

Example 4.5. *The Figure 4.5 presents an example where the filtering rule of Proposition 4.1 misses some arc-inconsistent values. The Figure 4.5a shows the precedence graph and the Figure 4.5b shows the variables' domains in the form of a value graph. The domains are consistent with the precedence constraints, as well as the* allDifferent *constraint with a generalized arc-consistency strength.*

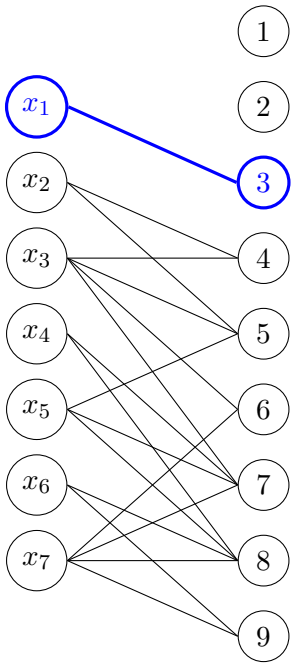
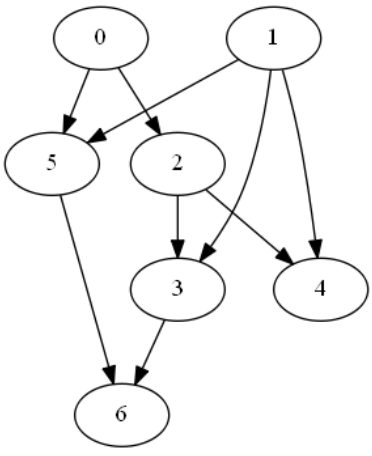


FIGURE 4.6: Representation of $G_{x_1 \to 3}$ corresponding of example depicted in Figure 4.5

The value 3 *should be removed from* $D(x_1)$*, which is why the edge is a dotted line. There exists no support for the assignment* $x_1 = 3$*, thus arc-consistency would remove the value. However, the filtering rule of Proposition 4.1 would not remove the value because there exists a matching of size* 7 *in* $G_{x_1 \to 3}$ *(which is shown in Figure 4.6):* $\{x_1 \to 3, x_2 \to 4, x_3 \to 5, x_4 \to 7, x_5 \to 8, x_6 \to 9, x_7 \to 6\}$*. This matching is obviously not a support because it would violate the precedence* $x_5 < x_7$*.*

4.6 Non-idempotency of each filtering scheme



(A) Precedence graph used to show non-idempotency

Variables	Domains
x_0	{ 0,1,2,3,5,6 }
x_1	{ 0,1,4,5,6,7 }
x_2	{ 0,3,5,6,7 }
x_3	{ 1,2,3,4,5,6,7 }
x_4	{ 0,1,2,3,4,5,7 }
x_5	{ 0,1,3,4,5,6,7 }
x_6	{ 1,2,3,4,5,7 }

(B) Variables' domains

FIGURE 4.7: Example of non-idempotency of each filtering scheme

In this section, we present an instance of the ALLDIFFPREC constraint where none of the algorithms presented above achieves idempotency in one run. The example is

depicted in Figure 4.7, and the state of the domains before and after running the filtering scheme an i^{th} time (the step in our tables) are given in Tables 4.1, 4.2, 4.3, 4.4 and 4.5.

Step	Before/After	$D(x_0)$	$D(x_1)$	$D(x_2)$	$D(x_3)$	$D(x_4)$	$D(x_5)$	$D(x_6)$
1	Before	{ 0,1,2,3,5,6 }	{ 0,1,4,5,6,7 }	{ 0,3,5,6,7 }	{ 1,2,3,4,5,6,7 }	{ 0,1,2,3,4,5,7 }	{ 0,1,3,4,5,6,7 }	{ 1,2,3,4,5,7 }
	After	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5,7 }	{ 1,3,4,5,6 }	{ 5,7 }
2	Before	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5,7 }	{ 1,3,4,5,6 }	{ 5,7 }
	After	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5,7 }	{ 3,4,5,6 }	{ 5,7 }
3	Before	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5,7 }	{ 3,4,5,6 }	{ 5,7 }
	After	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5,7 }	{ 4,5,6 }	{ 7 }
4	Before	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5,7 }	{ 4,5,6 }	{ 7 }
	After	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5 }	{ 4,5,6 }	{ 7 }

Table 4.1: States of domains at each step for BessiereEtAl

Step	Before/After	$D(x_0)$	$D(x_1)$	$D(x_2)$	$D(x_3)$	$D(x_4)$	$D(x_5)$	$D(x_6)$
1	Before	{ 0,1,2,3,5,6 }	{ 0,1,4,5,6,7 }	{ 0,3,5,6,7 }	{ 1,2,3,4,5,6,7 }	{ 0,1,2,3,4,5,7 }	{ 0,1,3,4,5,6,7 }	{ 1,2,3,4,5,7 }
	After	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5,7 }	{ 4,5,6 }	{ 7 }
2	Before	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5,7 }	{ 4,5,6 }	{ 7 }
	After	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5 }	{ 4,5,6 }	{ 7 }
3	Before	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5 }	{ 4,5,6 }	{ 7 }
	After	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5 }	{ 4,5,6 }	{ 7 }

Table 4.2: States of domains at each step for GreedyBC

Step	Before/After	$D(x_0)$	$D(x_1)$	$D(x_2)$	$D(x_3)$	$D(x_4)$	$D(x_5)$	$D(x_6)$
1	Before	{ 0,1,2,3,5,6 }	{ 0,1,4,5,6,7 }	{ 0,3,5,6,7 }	{ 1,2,3,4,5,6,7 }	{ 0,1,2,3,4,5,7 }	{ 0,1,3,4,5,6,7 }	{ 1,2,3,4,5,7 }
	After	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5,7 }	{ 4,5,6 }	{ 7 }
2	Before	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5,7 }	{ 4,5,6 }	{ 7 }
	After	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5 }	{ 4,5,6 }	{ 7 }
3	Before	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5 }	{ 4,5,6 }	{ 7 }
	After	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5 }	{ 4,5,6 }	{ 7 }

Table 4.3: States of domains at each step for GreedyRC

Step	Before/After	$D(x_0)$	$D(x_1)$	$D(x_2)$	$D(x_3)$	$D(x_4)$	$D(x_5)$	$D(x_6)$
1	Before	{ 0,1,2,3,5,6 }	{ 0,1,4,5,6,7 }	{ 0,3,5,6,7 }	{ 1,2,3,4,5,6,7 }	{ 0,1,2,3,4,5,7 }	{ 0,1,3,4,5,6,7 }	{ 1,2,3,4,5,7 }
	After	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5,7 }	{ 4,5,6 }	{ 7 }
2	Before	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5,7 }	{ 4,5,6 }	{ 7 }
	After	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5 }	{ 4,5,6 }	{ 7 }
3	Before	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5 }	{ 4,5,6 }	{ 7 }
	After	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5 }	{ 4,5,6 }	{ 7 }

Table 4.4: States of domains at each step for GodetBC

Step	Before/After	$D(x_0)$	$D(x_1)$	$D(x_2)$	$D(x_3)$	$D(x_4)$	$D(x_5)$	$D(x_6)$
1	Before	{ 0,1,2,3,5,6 }	{ 0,1,4,5,6,7 }	{ 0,3,5,6,7 }	{ 1,2,3,4,5,6,7 }	{ 0,1,2,3,4,5,7 }	{ 0,1,3,4,5,6,7 }	{ 1,2,3,4,5,7 }
	After	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5,7 }	{ 4,5,6 }	{ 7 }
2	Before	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5,7 }	{ 4,5,6 }	{ 7 }
	After	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5 }	{ 4,5,6 }	{ 7 }
3	Before	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5 }	{ 4,5,6 }	{ 7 }
	After	{ 0,1,2 }	{ 0,1 }	{ 3 }	{ 4,5,6 }	{ 4,5 }	{ 4,5,6 }	{ 7 }

Table 4.5: States of domains at each step for GodetRC

For GreedyBC, GreedyRC, GodetBC and GodetRC, the non-idempotency is due to the allDifferent constraint more especially and not really to the allDiffPrec constraint per say.

One should note that the non-idempotency of each of the presented algorithm is due to holes in domains. Whenever considered domains (whether domains after direct pruning or original domains) are interval domains, each of the presented filtering algorithm is idempotent with respect to bounds(Z) or range consistencies.

4.7 Comparison of filtering strength and prerequisites of algorithms

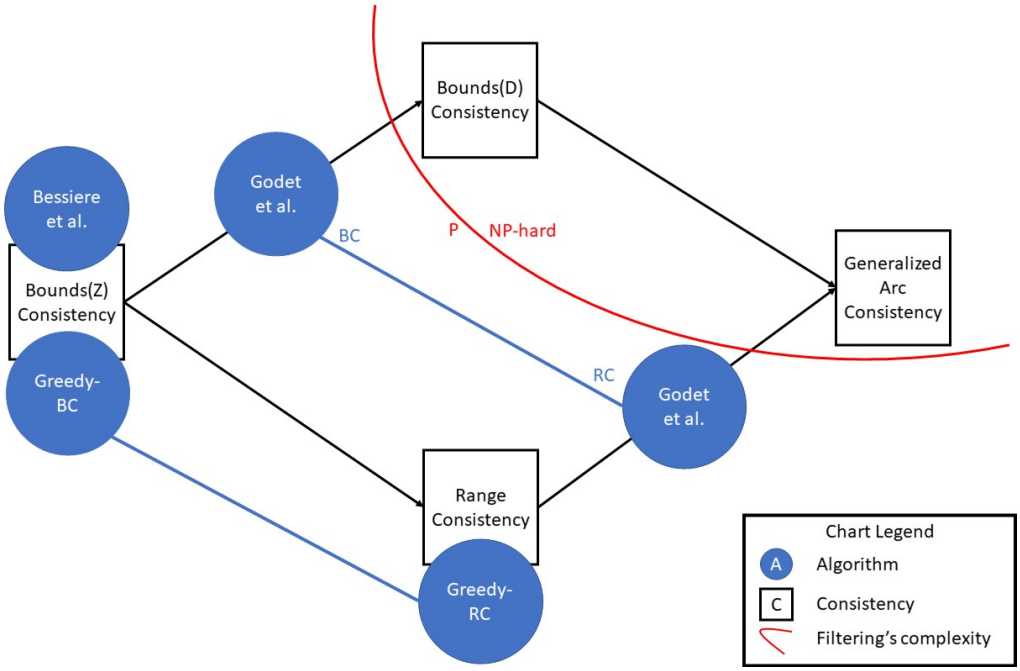


FIGURE 4.8: ALLDIFFPREC constraint's consistencies and algorithms

It is now interesting to discuss all the filtering strength of each algorithm. In Figure 4.8, we remind the comparison strength between bounds(Z), bounds(D), range and generalised-arc consistencies: bounds(Z) consistency is strictly weaker than range consistency and bounds(D) consistency, which are incomparable to one another and are both strictly weaker than generalized-arc consistency. The line in red reminds that computing support for the ALLDIFFPREC constraint is $\mathcal{NP}$-hard [Bes+11]. Each blue circle represents an implementation of the ALLDIFFPREC constraint presented earlier in this chapter.

The algorithm of Bessiere et al. [Bes+11], with ALLDIFFERENT and precedence constraints bounds(Z) consistent, is equivalent to bounds(Z) consistency for the ALLDIFFPREC constraint. From Theorem 4.1 and Example 4.3, we know that GODETBC is strictly stronger than bounds(Z) consistency, and from Theorem 4.2 and Example 4.3 that GODETRC is strictly stronger than range consistency. From Example 4.5 we also know that our filtering rule is strictly weaker than generalised-arc consistency, which is logical as our filtering rule has a polynomial complexity and Bessiere et al. prove that enforcing GAC is $\mathcal{NP}$-hard for the ALLDIFFPREC constraint.

Approach	Precedence constraints prior filtering	ALLDIFFERENT constraint prior filtering	Complexity
BESSIEREETAL [Bes+11]	Yes	Yes ($O(n)$ only once before execution)	$O(n(n+d))$
GREEDYBC	No	No	$O(n^3d^2)$
GODETBC	No	No	$O(n^3d^2 + n^2d^2\sqrt{n+d})$

TABLE 4.6: Comparison of requirements between BC(Z) algorithms

Table 4.6 and Table 4.7 give a summary of prior needed filtering and complexity of each implementation over one run. Precedence constraints are not needed for

Approach	Precedence constraints prior filtering	allDifferent constraint prior filtering	Complexity
GreedyRC	No	No	$O(n^3d^2)$
GodetRC	No	No	$O(n^3d^2 + n^2d^2\sqrt{n+d})$

Table 4.7: Comparison of requirements between RC algorithms

GreedyBC, GreedyRC, GodetBC and GodetRC as they are using domains after direct pruning, on which precedence are enforced before running the algorithm.

Note that the worst-case time complexity of our approach, as well as the one based on the greedy algorithm, are the same for BC(Z) and RC versions. However, in practice, we can expect that BC(Z) versions will be faster and should filter less than RC versions. Note also that the complexity of GodetBC can be made in $O(n^3d\log(d)+n^2d\log(d)\sqrt{n+d})$ and the one of GreedyBC in $O(n^3d\log(d))$, looking for bound support for bounds by dichotomy, but we did not indicate such complexities as it is not how it was encoded.

4.8 The GeneralizedAllDiffPrec constraint

The allDiffPrec constraint can be generalised to treat the precedence as variables.

Definition 4.4 (GeneralizedAllDiffPrec)**.** *Let $X = \{x_1, \dots, x_n\}$ be a set of n integer variables, and let $\hat{O} = \{\hat{o}_{i,j} \mid i \neq j \in [1,n]\}$ be $n(n-1)$ boolean variables representing the precedence. The constraint* GeneralizedAllDiffPrec *is defined as:*

$$\text{GeneralizedAllDiffPrec}(X, \hat{O}) \iff \begin{cases} \text{allDifferent}(X) \\ \forall i \neq j \in [1,n],\ \hat{o}_{i,j} \implies x_i < x_j \end{cases} \tag{4.7}$$

Notice that since the allDifferent constraint forces a total order on X, this constraints entails $\hat{o}_{i,j} = \neg\hat{o}_{j,i}$, for any two distinct integers i and j in $[1,n]$. Therefore, the implication in Equation 4.7 is in fact an *equivalence*.

This constraint can be useful for instance in scheduling problems to channel the start times of tasks to their relative positions, as we will see in Chapter 8.

It is easy to see that GeneralizedAllDiffPrec indeed generalises allDiffPrec. Given a n^2 boolean matrix O, let $\hat{O}$ denote the set of $n(n-1)$ boolean variables such that, for all distinct $i, j \in [1,n]$, $D(\hat{o}_{i,j}) = \{\texttt{true}\}$ if and only if $o_{i,j}$ is `true` and $D(\hat{o}_{i,j}) = \{\texttt{true}, \texttt{false}\}$ otherwise.

Lemma 4.4. *An assignment of X is a solution of* allDiffPrec(X, O) *if and only if it can be extended to a solution of* GeneralizedAllDiffPrec$(X, \hat{O})$.

Proof. It is easy to see that an assignment of X that can be extended to a solution of GeneralizedAllDiffPrec$(X, \hat{O})$ is a solution of allDiffPrec(X, O), since Equation 4.7 and Equation 4.1 are equal.

Conversely, consider a solution of allDiffPrec(X, O). Then, for all distinct $i, j \in [1,n]$, assign the variable $\hat{o}_{i,j}$ to `true` if $x_i < x_j$ and to `false` otherwise. Observe that this assignment is valid. Indeed, every boolean variable contains the value `true`, and $D(\hat{o}_{i,j})$ contains `false` unless $o_{i,j}$ is `true`. Moreover, it is consistent, again because Equation 4.7 and Equation 4.1 are equal. □

A corollary of Lemma 4.4 is that a (bound) support of GeneralizedAllDiffPrec$(X, \hat{O})$ containing the instantiation $x_i \to d$ exists if an

only if there is a (bound) support containing the same instantiation for the constraint allDiffPrec(X, O). In other words, when filtering the domains of the variables X, one should ignore all unassigned precedence variables, and apply the pruning from allDiffPrec with the weakest possible partial order O, since (bound) supports can always be extended to the variables O.

However, it may be possible to prune the domain of variables in $\hat{O}$, and hence infer new precedence, as shown in Example 4.6.

Example 4.6. *Let* $D(x_1) = \{1, 2, 3\}, D(x_2) = \{2, 3, 4, 5, 6\}, D(x_3) = D(x_4) = D(x_5) = \{2, 3, 4, 5\}$. *Moreover, let* O *be the boolean matrix where* $o_{1,3} = o_{1,4} =$ `true` *and* $o_{i,j} =$ `false` *otherwise. In such a case,* $D(\hat{o}_{1,3}) = D(\hat{o}_{1,4}) = \{$`true`$\}$ *and* $D(\hat{o}_{i,j}) = \{$`true`, `false`$\}$ *otherwise.*

It can be verified that the constraint allDiffPrec(X, O) *is arc consistent, however, the instantiation* $\hat{o}_{1,2} \to 0$ *is not bounds consistent for* GeneralizedAllDiffPrec$(X, \hat{O})$.

Moreover, we must maintain the transitivity of the precedence graph given by the variables in $\hat{O}$, *i.e.* the following constraint is implied by GeneralizedAllDiffPrec$(X, \hat{O})$:

Definition 4.5 (transitivity)**.** *Let* $\hat{O} = \{\hat{o}_{i,j} \mid i \neq j \in [1, n]\}$ *be* $n(n-1)$ *Boolean variables.*

$$\text{transitivity}(\hat{O}) \iff \forall i \neq j \neq k \in [1, n],\ \hat{o}_{i,j} \wedge \hat{o}_{j,k} \implies \hat{o}_{i,k} \tag{4.8}$$

Therefore, in order to achieve bounds(Z) consistency on GeneralizedAllDiffPrec$(X, \hat{O})$, we must achieve bounds(Z) consistency on transitivity$(\hat{O})$. Observe, however, that transitivity pruning does not invalidate Lemma 4.4 since a bound support satisfies all precedence if and only if it satisfies their transitive closure.

Lemma 4.5. *If the domains* $D_{x_i \to d_i}$ *and* $D_{x_j \to d_j}$ *are bounds(Z) consistent for the constraint* allDifferent(X) *but* $D_{x_i \to d_i, x_j \to d_j}$ *is unsatisfiable, then there exists an interval* $[a, b]$ *that contains* d_i, d_j *and the domains of* $b - a$ *variables other than* x_i *and* x_j.

Proof. Since the domain $D_{x_i \to d_i, x_j \to d_j}$ has no solution, there exists an interval $[a, b]$ containing the domains of at least $b - a + 2$ variables. Suppose that $d_i > b$, then the same interval exists in $D_{x_j \to d_j}$, which contradicts the hypothesis. A similar argument can be made for $d_i < a$, $d_j > b$ and $d_j < a$. □

Lemma 4.6. *There is a bound support of* GeneralizedAllDiffPrec$(X, \hat{O})$ *containing the instantiation* $\hat{o}_{i,j} \to$ `true` *if and only if after enforcing bounds(Z) consistency on* allDifferent(X) *with domain* $D_{x_i \to \underline{x_i}}$, *we have* $\overline{x_j} > \underline{x_i}$.

Proof. First, observe that the following weaker claim holds:

There is a bound support of GeneralizedAllDiffPrec$(X, \hat{O})$ containing the instantiation $\hat{o}_{i,j} \to$ `true` if and only if there exists $d \in D(x_i)$ such that after enforcing bounds(Z) consistency on allDifferent(X) with domain $D_{x_i \to d}$, we have $\overline{x_j} > d$.

- *If (⇐):* Since after enforcing bounds(Z) consistency on allDifferent(X) with domain $D_{x_i \to d}$ the upper bound of x_j is $\overline{x_j}$, then there exists a bound support of allDifferent(X) with domain $D_{x_i \to d, x_j \to \overline{x_j}}$.

Moreover, by Lemma 4.3 this assignment can be turned into a bound support of allDiffPrec(X, O) with domain $D_{x_i \to d, x_j \to \overline{x_j}}$.

Finally, by Lemma 4.4 we know that this assignment can be extended to a solution of GeneralizedAllDiffPrec$(X, \hat{O})$, with $\hat{o}_{i,j} \to \texttt{true}$, since $x_i < x_j$ as $d < \overline{x_j}$.

- *Only If ($\Rightarrow$):* Suppose that after enforcing bounds consistency on allDifferent(X) with domain $D_{x_i \to d}$, we have $\overline{x_j} < d$.

  Then allDifferent(X) is unsatisfiable for the domain $D_{x_i \to d, x_j \to d'}$ for any $d' > d \in D(x_j)$. As allDifferent is a specialization of allDiffPrec, if allDifferent(X) is unsatisfiable, so is allDiffPrec(X, O).

  Therefore, a bound support of allDifferent(X) with domain $D_{x_i \to d}$ may not be extended to GeneralizedAllDiffPrec$(X, \hat{O})$ containing $\hat{o}_{i,j} \to \texttt{true}$ (since necessarily $x_i > x_j$).

Now, suppose that the claim is true for $d > \underline{x_i}$ but is false for $\underline{x_i}$. Let u be $\overline{x_j}$ after enforcing bounds consistency on allDifferent(X) with domain $D_{x_i \to d}$. We have $\underline{x_i} < d < u$. Observe that the domains $D_{x_i \to \underline{x_i}}$ and $D_{x_j \to u}$ have a bound support for allDifferent(X), which is unsatisfiable with domain $D_{x_i \to \underline{x_i}, x_j \to u}$. By Lemma 4.5, there exist $a \leq \underline{x_i}$ and $b \geq u$ such that $[a, b]$ contains the domains of $b - a$ variables other than x_i and x_j. Therefore, if x_i and x_j are assigned respectively to d and u, then the interval $[a, b]$ contains the domains of $b - a + 2$ variables, and hence allDifferent(X) is unsatisfiable with domain $D_{x_i \to d, x_j \to u}$. This contradicts the hypothesis and therefore proves the Lemma. □

Theorem 4.4. *Bounds consistency on* GeneralizedAllDiffPrec$(X, \hat{O})$ *can be achieved in* $O(n^3)$ *down a branch of the search tree.*

Proof. First, bounds(Z) consistency on the constraint transitivity$(\hat{O})$ can be achieved in $O(n^2)$ amortised time. In order to do that, we need to maintain the graph of precedence and use an incremental algorithm. For instance, the algorithm proposed by Ibaraki and Kato [IK83] takes $O(n^3)$ in total (i.e., over a branch of the search tree), and hence $O(n^2)$ time amortized in a branch of length n.

Then, using the algorithm from [Bes+11], bounds consistency on allDiffPrec(X, O) can be achieved in $O(n^2)$ time. By Lemma 4.4, every bound support can be extended to the variables $\hat{O}$, hence we know that the domains of variables in X cannot be reduced further.

Next, we use Lemma 4.6 to prune the domains of the variables in $\hat{O}$. Bounds(Z) consistency on allDifferent(X) can be achieved in $O(n)$ plus the time to sort the variables twice (by their lower and upper bounds) [MT00]. We run this algorithm $2n$ times, that is, on the domains $D_{x_i \to \underline{x_i}}$ and $D_{x_i \to \overline{x_i}}$ for every variable x_i. However, sorting the variables can be done in linear time at each iteration: suppose that the variables $x_1, \ldots, x_n$ are sorted by non-decreasing lower bounds, and consider any variable x_i. First, all variables have the same lower bound in domains $D_{x_i \to \underline{x_i}}$, hence the ordering is unchanged. Now, consider the domains $D_{x_i \to \overline{x_i}}$. The variables in $\{x_1, \ldots, x_{i-1}\}$ have the same lower bound in $D_{x_i \to \overline{x_i}}$ and hence do not need to be sorted again. Moreover, the variables in $\{x_{i+1}, \ldots, x_n\}$ that are not successors of x_i according to the partial order $\hat{O}$ also have the same lower bounds and hence are still pairwise sorted. The variables in $\{x_{i+1}, \ldots, x_n\}$ that are successors of x_i according to the partial order $\hat{O}$ have for lower bound the maximum between their previous lower bounds and $\overline{x_i} + 1$. Therefore, they are still pairwise sorted. The two latter lists can

be merged in linear time, hence we can sort the domains $D_{x_i \to \underline{x_i}}(x_1), \ldots, D_{x_i \to \underline{x_i}}(x_n)$ by lower bounds in $O(n)$.

Now, we compute two sorted lists of the variables in X, by non-decreasing lower and upper bounds in $O(n \log n)$ time, and for every $i \in [1, n-1]$:

- We "repair" the sort on each bound as shown above, and we enforce bounds(Z) consistency on ALLDIFFERENT(X) for the domains $D_{x_i \to \underline{x_i}}$ and $D_{x_i \to \overline{x_i}}$ in $O(n)$ time.
- For every $j \in [i+1, n]$, we use Lemma 4.6 to prune $\hat{o}_{i,j}$ (and $\hat{o}_{j,i}$): if $\overline{x_i} < \underline{x_j}$ then $\hat{o}_{i,j} \to \texttt{false}$ and $\hat{o}_{j,i} \to \texttt{true}$ are bounds inconsistent. Similarly, if $\underline{x_i} > \overline{x_j}$ then $\hat{o}_{i,j} \to \texttt{true}$ and $\hat{o}_{j,i} \to \texttt{false}$ are bounds inconsistent. This can be done in $O(n)$ time.

The overall worst-case time complexity is therefore in $O(n^2)$ amortized: $O(n^2)$ amortized for the transitivity; $O(n^2)$ to achieve bounds consistency on ALLDIFFPREC(X, O); $O(n \log n)$ for the two initial sorts; and $O(n)$ for each of the $O(n)$ iterations of the loop. In the worst case, it is $O(n^3)$ for the transitivity plus $O(n^2)$ for the rest. □

4.9 Conclusion and future works

In this chapter, we gave a complete overview of the ALLDIFFPREC constraint, introduced by Bessiere et al. [Bes+11] as a global constraint combining an ALLDIFFERENT and precedence constraints. We discussed on state-of-the-art results and most especially on the worst-case time complexity of the filtering scheme based on the greedy bound support building algorithm. We gave a correction of the state-of-the-art algorithm and show how to build a similar one based on the same idea. Then we introduced a new filtering rule that considers potential holes within domains. We showed that this new rule leads to consistencies strictly greater than bounds(Z) and range. Finally, we summarised the prior filtering and worst-case time complexities of each filtering scheme.

Future works could be oriented to improve on the global complexity of the filtering scheme of GODETBC. Indeed, it might be possible to decrease the worst-case time complexity of the approach as Bessiere et al. did from the greedy bound support building algorithm to their algorithm. Focusing on incrementality between runs might be a way.

Chapter 5

Implementing the allDiffPrec constraint in a constraint solver

Implementation is often an overlooked part when developing new constraints and propagators. However, paying attention on the code development as well as profiling how and where CPU time is spent and trying to improve on this might lead to better performance in practice. In this chapter, we discuss on implementation details of the different configurations of the allDiffPrec constraint that were presented in Chapter 4.

The chapter is organised in two sections. In Section 5.1, we discuss on implementation details of each filtering scheme within a constraint solver. We give insights on which data structures to use and how to implement them, as well as how to organise the code of the propagator in order to improve execution speed of each filtering scheme. In Section 5.2, we present an experimental protocol to compare the different filtering schemes and discuss on the obtained results.

The work presented in this chapter was submitted to the CP conference 2021.

5.1 Implementation Schema

In this section, we will discuss about implementation details of each version of the allDiffPrec constraint that we discussed all through the previous chapter. Paying attention to the implementation of all the algorithms lead to better performance, whereas simple implementations can be quite slow in practice. Most especially, great attention should be given to the data structures and the efficiency of operations on them.

Let us remind that, in this thesis, we concentrate on Choco solver [PFL17]. As such, some remarks and developments discussed in this section might not be true for all constraint solvers, as they do not all rely on the same internal mechanisms. Most especially, Choco solver relies on the use of lists of priorities for propagators, as presented in Paragraph 5 in Section 2.2.5.

5.1.1 Propagator structure

We can first see from Table 4.6 and Table 4.7 that, whatever the implementation of the allDiffPrec constraint, prior filtering are supposed to be done before running the filtering algorithm associated with the implementation of the constraint. And when it is not the case, doing so might lead to better performance in practice (as we will see later). Precedence constraints are supposed to be bounds(Z) consistent

before running any of the filtering algorithm. For the implementation BessiereEtAl, the allDifferent constraint should also be bounds(Z) consistent before running the filtering algorithm (Algorithm 4.5), let it be for lower bounds or upper bounds filtering.

Moreover, a particular attention should be given to the updates of variables to assure that the domains stay consistent with respect to the precedence constraints, as well as the allDifferent constraint in the case of the implementation BessiereEtAl.

Resting only on the internal mechanisms of the solver to guarantee these prior filtering might be difficult or very inefficient. More especially, in Choco solver, propagators are supposed to be idempotent by the scheduling module, which is in charge of the priority lists and of calling the propagators that need to be ran. It is possible to manually schedule a propagator, however the system is not conceived for this and later developments in the solver might break such a functioning. To avoid such worries, the propagator for the allDiffPrec constraint can manage itself the prior filtering of precedence constraints and the allDifferent if needed.

Algorithm 5.1: Propagate function for the propagator for allDiffPrec constraint

```
1  Function Propagate(X : variables, O : precedence set)
2  begin
3      do
4          do
5              priorFilt ← updateBound(true)
6              priorFilt ← priorFilt ∨ updateBound(false)
7              if filtAllDifferent then
8                  priorFilt ← priorFilt ∨ allDifferent.filter()
9              end
10         while priorFilt
11     while Filter(X, O)
12 end
```

Algorithm 5.1 presents the implementation that was done. The function updateBound is in charge of filtering the precedence constraints, filtering the lower bounds whenever the parameter is `true` and filtering the upper bounds whenever the parameter is `false`. The function returns `true` whenever one bound has been filtered during the procedure, and `false` otherwise. Let us note that, for a better efficiency of this function, the precedence graph induced by O should be travelled following the topological order. $filtAllDifferent$ is a parameter of the propagator that is `true` for BessiereEtAl and `false` for all other implementations. If $filtAllDifferent$ is `true`, then the allDifferent constraint is filtered with bounds(Z) consistency using the filtering algorithm of Mehlhorn et al. [MT00]. Lines 4 to 10 are ran again and again until a fixpoint is reached. Whenever it is the case, the function Filter() is called, which ran the filtering algorithm associated with the wanted implementation between GreedyBC, GreedyRC, BessiereEtAl, GodetBC or GodetRC.

In the following subsections, we are going to discuss the different implementation of the function Filter(X, O) according to each filtering algorithm discussed in the previous chapter. For convenience, we do not necessarily indicate all data structures within parameters of functions, as it is up to the developers to rely on Object Oriented

Programming or not, and as such how data structures are shared between the different part of the code. For instance, the topological order to travel across the precedence graph is computed once and for all at the creation of the propagator and is stored in it, being shared to dedicated objects in charge of the filtering algorithms per say when needed.

5.1.2 Implementation details for GreedyBC and GreedyRC

The computing of $m = \min_{\substack{x_j \in X \\ |D(x_j)|>1}} \min(D_\alpha^{dp}(x_j))$, $M = \max_{\substack{x_j \in X \\ |D(x_j)|>1}} \max(D_\alpha^{dp}(x_j))$ and $\mathcal{D} \leftarrow \bigcup_{\substack{x_j \in X \\ |D(x_j)|=1}} D(x_j)$ can be done once and for all before running the algorithm. Whenever running Algorithm 4.3 leads to the instantiation of a variable, $\mathcal{D}$ can be updated.

Building the domains after pruning $D_\alpha^{dp}(x_1), \ldots, D_\alpha^{dp}(x_n)$ when looking for a bound support for $x_v \to \alpha$ is done quite easily. In our case, we stored the minimal and maximal values of variables in integer arrays to manipulate domains after pruning without changing the true domains, unless a removal should happen. Computing the domains $D_\alpha^{dp}(x_1), \ldots, D_\alpha^{dp}(x_n)$, without enforcing precedence constraints on them, can be done in $O(n)$.

To accelerate the preprocessing of the domains after direct pruning, one should travel the precedence graph following the topological order when updating lower bounds and the inverse of the topological order when updating upper bounds of domains after direct pruning. This way, all domains after direct pruning are consistent with precedence constraints in $O(n^2)$. Thanks to the phase of preprocessing the domains after direct pruning, variables can be updated as soon as a bound update is detected, which can trigger a failure earlier if a domain is emptied.

Execution speed can be earned when adding available variables (variables whose minimal value in domains after direct pruning is the current treated value v): when looping over variables to see if they should be added to the list of available variables or not, the next value for which the procedure of adding available variables should be ran again can be computed during the loop as it is the smallest value strictly greater than v among all minimal values of domains after direct pruning for non-assigned variables. When there are ranges of values v that are not minimal values of domains after direct pruning, the loop of adding available variables (which is in $O(n)$) is avoided all through the range.

Finally, let us discuss on the implementation of Greedy 1, Greedy 2 and Greedy 3 as introduced in Section 4.3. The implementation details on the domains after direct pruning are true for all these three implementations as they all use these domains. As discussed in Section 4.3, the distinction is after the build and filtering of the domains after direct pruning. These three implementations were compared on a subset of the benchmark composed of 30 instances of the allDiffPrec constraint of size $n = 50$ (see Section 5.2) for which GreedyBC (using Greedy 1 implementation) took more than $1s$ to do the optimality proof. The idea was to compare the speed of execution of each implementation on instances for which the time to proof was not too short to avoid the experience to be influenced by the current charge of the processor. On these 30 instances, Greedy 3 was 10% quicker than Greedy 1 in average and Greedy 2 was 12% quicker than Greedy 1 in average. As such, and even if it was not the case for our own experiments, if one wants to implement the allDiffPrec constraint

using the greedy filtering scheme, Greedy 2 or Greedy 3 implementations should be favoured. Note that implementing Greedy 3 is not very hard if the bounds(Z) consistency filtering algorithm for allDifferent [Lóp+03] is already implemented within the constraint solver. Greedy 2 is in any case very easy to implement.

Despite it is not necessary that precedence constraints are bounds(Z) consistent prior to the execution of the filtering algorithm, it greatly improves execution speed in practice when they have been enforced bounds(Z) consistent. Indeed, it will avoid to run the filtering algorithm, with the build and filtering of domains after direct pruning, on instantiations $x_v \to \alpha$ that would have been filtered by precedence constraints.

5.1.3 Implementation details for BessiereEtAl

First, let us discuss about the UnionFind data structure. Let us first note that Tarjan proves in [Tar75] that the worst-case time complexity of m intermixed operations Find and Union is $O(m\alpha(n))$, where $\alpha(\cdot)$ is the inverse Ackermann function, for the disjoint-set forest implementation. As such, Find and Union operations have almost near-constant time complexity. Whenever the values stored in the UnionFind data structure are known in advance, Gabow and Tarhan presents a way to reduce the complexity of these operations to $O(1)$ [GT85]. However, these operations are not easy to implement and the constant time complexities are, in fact, constant only in specific cases. Let us also note that state-of-the-art implementations of the near-constant time complexities for Find and Union operations appear to be really efficient in practice. As such, we used this version of the UnionFind data structure.

Concerning Algorithm 4.5, the implementation is pretty straightforward. It quite follows the algorithm. Before running the algorithm, the lower and upper bounds (or their mirror values) are stored within integer arrays and these values are used all along the algorithm to avoid violating precedence constraints or allDifferent constraint whenever a bound update should be detected. Instead, the new upper bounds (or lower bounds for the mirror version) are stored in an integer array and updates on variables is done at the end of the algorithm. Whenever a bound update has been done at the end of the algorithm, whether for its lower or upper bounds version, the function Filter() immediately returns `true` for precedence constraints and the allDifferent constraint to be made bounds(Z) consistent afresh. If no bound update is detected, Filter() returns `false`.

5.1.4 Implementation details for GodetBC and GodetRC

The implementations of GodetBC and GodetRC are similar to the ones of GreedyBC and GreedyRC. The filtering scheme is based on Algorithm 4.1 for GodetBC and on Algorithm 4.2 for GodetRC. The hasBoundSupport function for variable x_v and value $\alpha \in D(x_v)$ consists in building the bipartite graph $G_{x_v \to \alpha}$ based on domains after direct pruning authorising holes, enforce precedence constraints on it, and finally compute the size of the maximum matching over it.

For efficiency, the bipartite graph $G_{x_v \to \alpha}$ should be modified incrementally. As such, the value graph is first built, and at each call to hasBoundSupport edge removals are stored in order to restore them at the end of the function. This implementation is faster than building from scratch the bipartite graph $G_{x_v \to \alpha}$ at each call to hasBoundSupport. For efficiency, for a given value w, it is useful to access rapidly to the smallest value strictly greater than w in the domain of a variable, and similarly to the

greatest value strictly smaller than w in the domain of a variable. In Choco solver, we use a DirectedGraph into which, for each node, sets of predecessors and successors are maintained using BitSet data structures, which allows for quick accessing next or previous value of w (we will suppose that this operation is in $O(1)$, but it is near-constant in reality as the operation depends on the number of bits used to represent the size of the BitSet). All in all, building the bipartite graph $G_{x_v \to \alpha}$ without enforcing the precedence constraints on it can be done in $O(nd)$.

Once the bipartite graph $G_{x_v \to \alpha}$ is built from the value graph, following the topological traversal of the precedence graph, $G_{x_v \to \alpha}$ is updated for domains after direct pruning $D_{x_v \to \alpha}$ to be preprocessed. As for GreedyBC and GreedyRC, using the topological traversal of the precedence graph achieves the preprocessing of domains after direct pruning $D_{x_v \to \alpha}$ in $O(n^2 d)$. It is not in $O(n^2)$ as for GreedyBC and GreedyRC as updating the bound of a domain after direct pruning is in $O(d)$ because of holes in the domains while GreedyBC and GreedyRC consider domains as intervals.

Finally, the maximum matching is computed. For efficiency, and to avoid to compute some augmenting paths, we first build a greedy matching by affecting to each variable their smallest available value. Thanks to the BitSet implementations for sets in the DirectedGraph, this can be done in $O(n)$. If after this greedy matching procedure, all variables have been matched, there is no need to look for augmenting paths, as a maximum matching of size n has already been found, and the value α should not be filtered from $D(x_v)$.

Attention should really be given to the building of $G_{x_v \to \alpha}$ as, in practice, it is what takes the most time. With attention on the implementation, we achieve building $G_{x_v \to \alpha}$ almost 5 times faster than the implementation building it from scratch.

As for GreedyBC and GreedyRC, it is not necessary that precedence constraints have been enforced with bounds(Z) consistency prior to the execution of the filtering algorithm. However, these prior filtering can avoid to spend time on instantiations $x_v \to \alpha$ that would have been filtered by precedence constraints. More precisely, doing these prior filtering improves on the execution speed by 10% for instances of $n = 50$ and up to 30% for instances of size $n = 200$.

5.2 Experiments

5.2.1 Experimental protocol

In this section, we will present an experimental protocol to compare the performance of the different versions of the allDiffPrec constraint.

5.2.1.1 Model for comparison

The comparison will be done on the Constraint Optimization Problem described in Figure 5.1, which consists of an allDiffPrec constraint and of the minimization of the maximum value taken by X variables.

As a base comparison, we wanted to compare to a decomposition of the allDiffPrec constraint. For this, we use the allDifferent constraint as well

$$\textit{minimise } z = \max_{x_v \in X} x_v \quad \textit{s.t.}$$

$$\textsc{allDiffPrec}(X, O) \tag{5.1}$$

Figure 5.1: Equations for allDiffPrec constraints description

as precedence constraints, all with bounds(Z) consistency. This model is referred to as Decomposition.

We will compare this base-version of allDiffPrec with the ones described earlier and summarized in Table 4.6 and Table 4.7: GreedyBC, GreedyRC, BessiereEtAl, GodetBC and GodetRC.

In order to compare the filtering algorithms' strength and speed of execution, we use a non-dynamic search whose branching scheme is the same for all models: inputOrderLb. It selects the first variable $x_i \in X$ that is not instantiated and branch on it by instantiating it to its lower bound ($x_i \in X$ s.t. $i = \min(\{j \in [1, n] \mid |D(x_j)| > 1\})$). This way, the search tree for each model should be very similar, the solving difference being the speed of execution of each model and the search tree is smaller for better consistencies.

5.2.1.2 Instance generation

All versions of allDiffPrec but GodetBC and GodetRC only use the bounds of the variables, while these two consider holes in the domains of the variables. As such, we expect that GodetBC and GodetRC versions should be better than the other four when the density (the percentage of values between the lower and upper bounds) of the domains of the variables is neither high (too close to true bounds(Z) consistency) nor low (a few filtering leads to instantiation or fail).

In order to confirm this hypothesis, we randomly generate instances. Given a density, all variables will have this density. Precedence O are randomly created with either 20%, 40%, 60% or 80% probability for each $(i, j) \in [1, n]^2$, with $i < j$, for x_i to precede x_j.

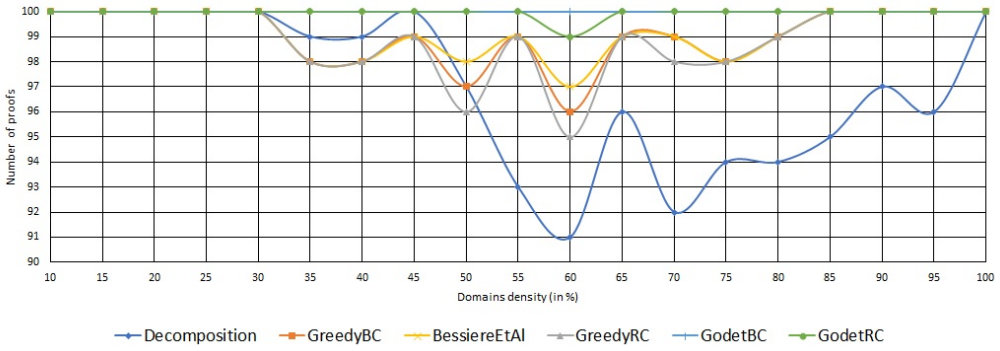
We generate 25 instances for each precedence probability and each domains density between 10% and 100% by 5%. In total, it leads to 1900 instances, 100 for each density. We do this for several sizes n of instances: $50, 100, 150$ and 200.

5.2.1.3 Execution environment

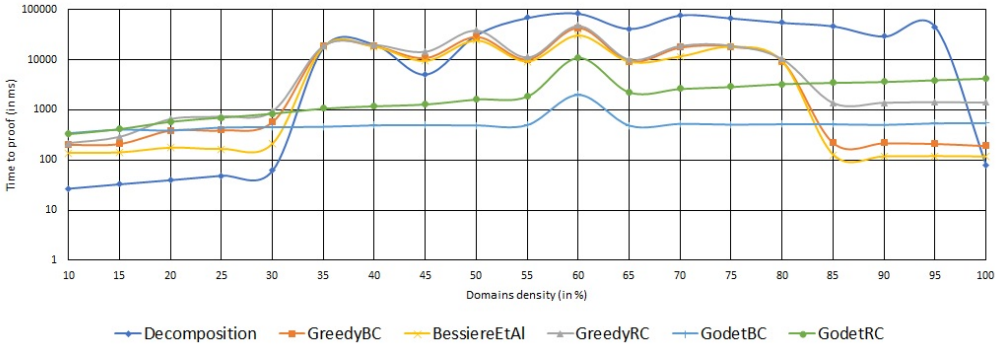
The Choco-solver library [PFL17] was used in its 4.10.5 version. A time limit of 15 minutes was given for each model and each instance. Every instance and piece of code that were used can be found in a GitHub repository [God21c].

5.2.2 Experimental results

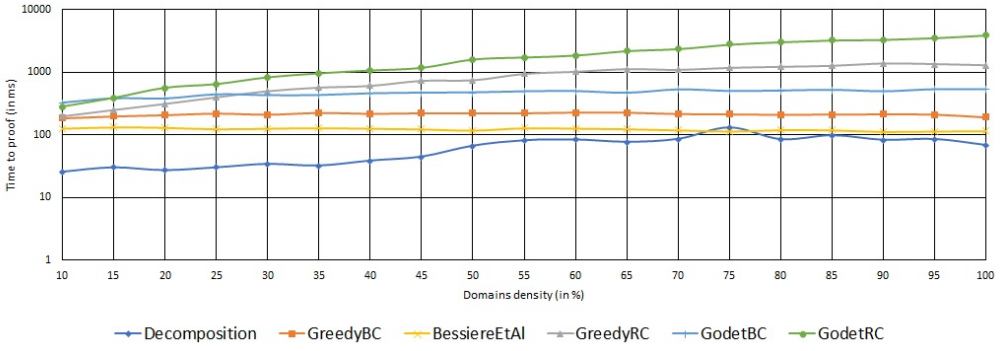
First, note that a benchmark of 1900 instances of size $n = 20$ was also generated. However, all configurations did the proof of optimality or absence of solution on all the 1900 instances, and the Decomposition was the quicker overall in average. This is logical as the number of explored nodes by the Decomposition configuration (which has the weaker filtering, not even enforcing the bounds(Z) consistency [Bes+11]) was in average not high enough, such that the high speed of execution of the configuration



(A) Number of proofs per density for each approach



(B) Mean time to proof per density for each approach

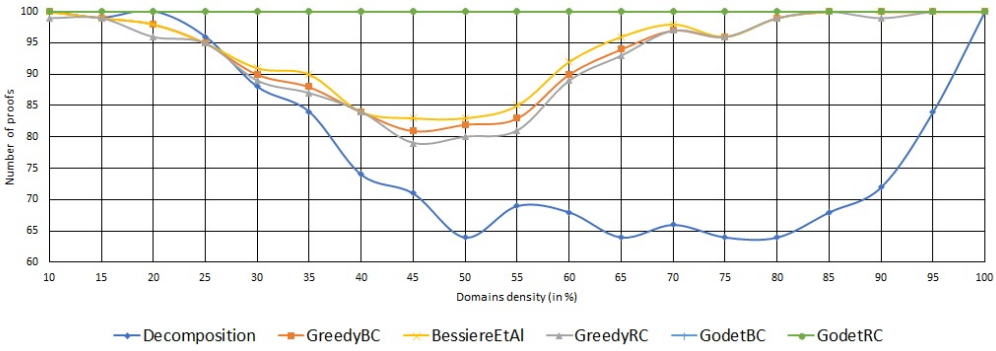


(C) Median time to proof per density for each approach

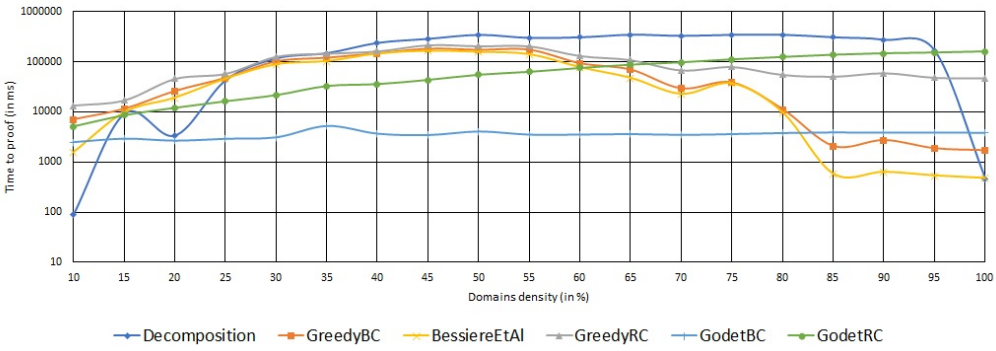
FIGURE 5.2: Results for instances of size $n = 50$

would largely compensate for the lack of filtering.

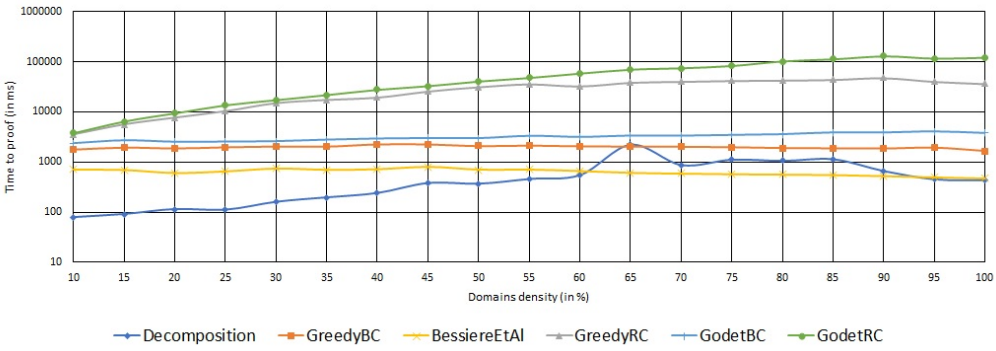
The results are given in Figure 5.2 for instances of size $n = 50$, in Figure 5.3 for instances of size $n = 100$, in Figure 5.4 for instances of size $n = 150$ and in Figure 5.5 for instances of size $n = 200$. In each figure and each graphic, the density of the domains varies on the abscissa axis. We remind that for each size and density, we generated 100 instances. In each figure, the first graphic indicates the number of proofs that were done by each configuration in function of the density of the domains, the second one indicates the mean time to proof in function of the density of the domains of each configuration, the third one indicates the median time to proof in function of the density of the domains of each configuration. For the mean and median times to proof, we included when the proof was not done: the considered value is, in such



(A) Number of proofs per density for each approach



(B) Mean time to proof per density for each approach

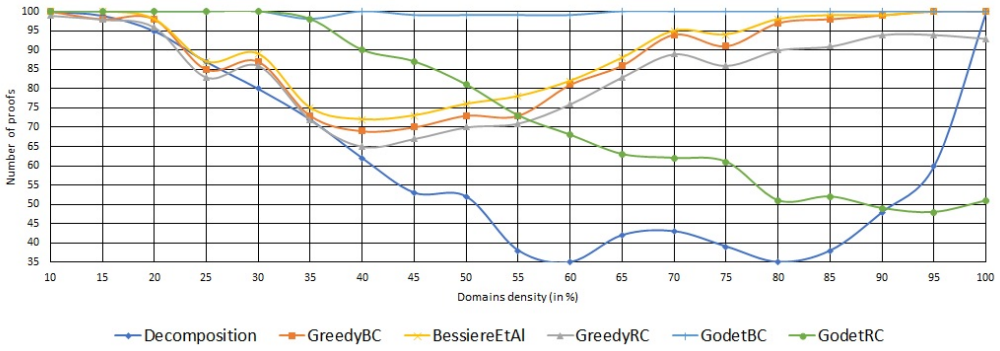


(C) Median time to proof per density for each approach

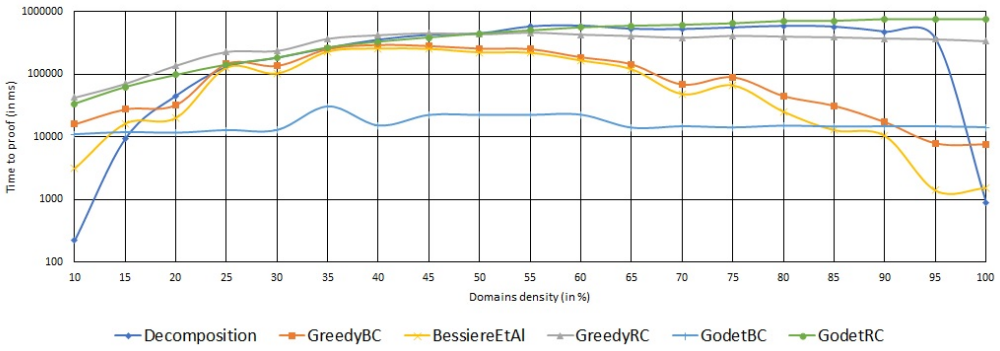
Figure 5.3: Results for instances of size $n = 100$

a case, the time limit. Therefore, and more especially for the mean time to proof, the configurations that miss to make the proof are not so many disadvantaged in the results.

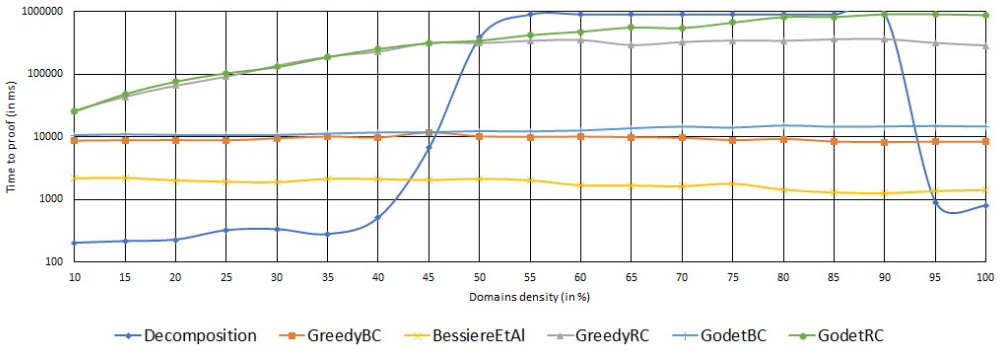
First, we can note that, whatever the size of the instances, the median time to proof graphics show that a majority of instances are not very difficult to solve, and therefore speed of execution prevails over filtering strength. We can temper this conclusion when observing the median time to proof for instances of size $n = 150$ and $n = 200$, where Decomposition completely falls behind for densities between 45% and 90%. Therefore, the BessiereEtAl implementation of the allDiffPrec constraint seems the most appropriate in most cases.



(A) Number of proofs per density for each approach



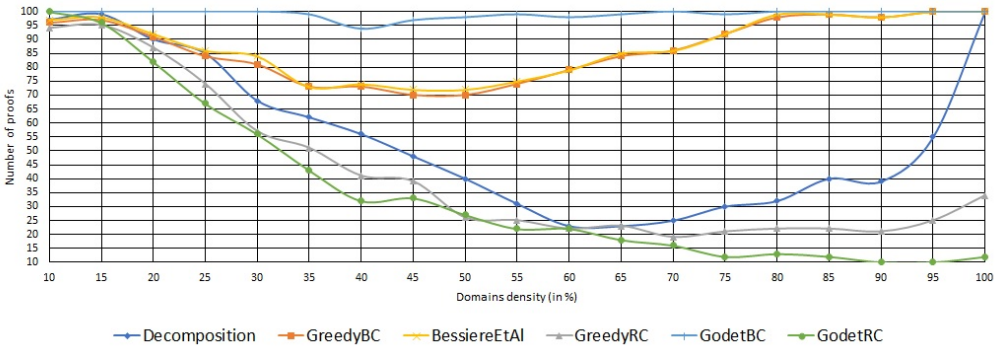
(B) Mean time to proof per density for each approach



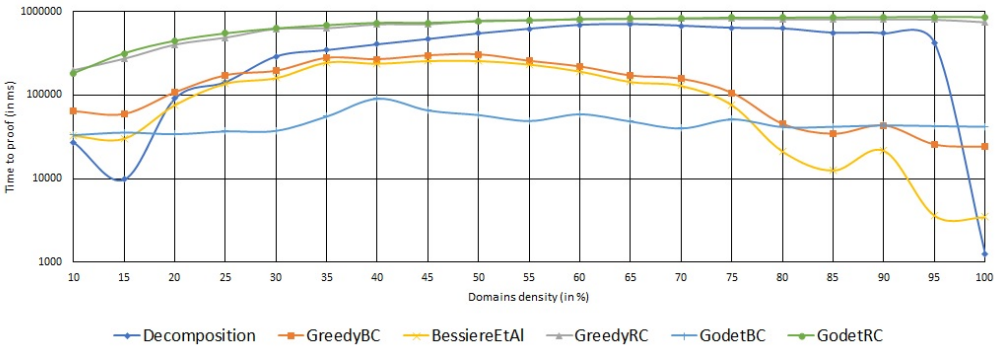
(C) Median time to proof per density for each approach

FIGURE 5.4: Results for instances of size $n = 150$

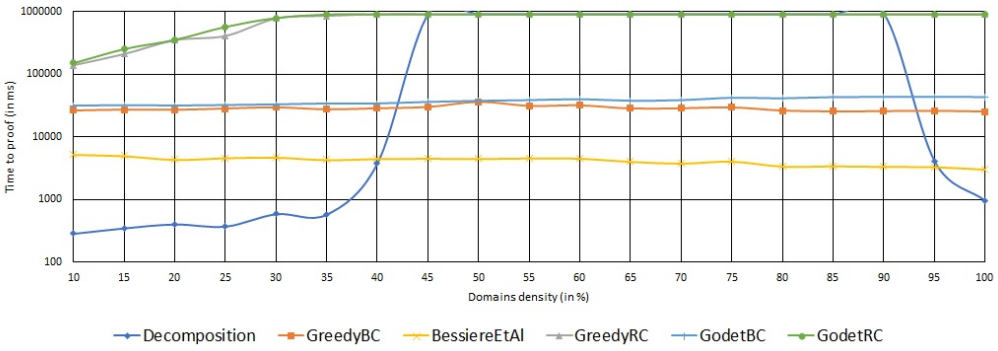
Notwithstanding, the graphics of the number of proofs and of the mean time to proofs both show that the BESSIEREETAL configuration faces challenges to do the proof when the density is between 20% and 75%. Note that failing to do the proof automatically degrades the mean time to proof, which is why graphics indicating the number of proofs and the mean time to proof generally have similar shapes. As we said, we expected all configurations except GODETBC and GODETRC to face difficulties to do the proof when the density of the domains of the variables is neither high (too close to true bounds(Z) consistency) nor low (a few filtering leads to instantiation or fail). The experimental results completely validate this intuition, the higher the size n the clearer the validation. As the curves of GREEDYBC, BESSIEREETAL and GREEDYRC are close to each other (except when $n = 200$ where GREEDYRC completely fall behind but we will discuss on this later), and we confirm it with the



(A) Number of proofs per density for each approach



(B) Mean time to proof per density for each approach



(C) Median time to proof per density for each approach

FIGURE 5.5: Results for instances of size $n = 200$

raw results data, they face difficulties to do the proof on the same instances and only speed of execution make a difference between these configurations.

One can easily see the great performance of our approach. Indeed, for instances of size $n = 50$ and $n = 100$, both GodetBC and GodetRC configurations do the proof on all the instances within the time limit. Despite having the same worst-case time complexity, GodetBC is quicker than GodetRC in average to do the proof. Let us remark that GodetBC maintains a high ratio of done proofs when the size of instances gets higher, where GodetRC seems to scale quite badly. Another remark we can make on GodetBC, when observing median time to proof graphics, is that it gets close results with GreedyBC. This can be explained by the fact that both

configurations are iteratively looking if an instantiation is inconsistent or not using domains after direct pruning. It seems that it is the build and filtering of precedence on these domains that take time, which is also what we can monitor in terms of CPU time.

When looking closer, both RC configurations (GreedyRC and GodetRC) face difficulties when the size of instances gets greater. If for instances of size $n = 150$, GreedyRC seems to keep close performance from GreedyBC, both GreedyRC and GodetRC completely fall behind on instances of size $n = 200$. Interestingly, one can remark that the greater the density becomes, the more difficulties both configurations are facing. This can be explained by the fact that d gets greater as the density gets greater, and therefore RC configurations fall behind because of their worst-case time complexity greatly depending on d. Both GreedyBC and GodetBC, whose worst-case time complexities also depend on d, do not fall behind because the algorithms are not run d times, which would correspond to an instantiation or a fail of the filtered variable.

5.3 Conclusion

In this chapter, we discussed the implementation within a constraint solver of the different configurations of the allDiffPrec constraint with execution speed in mind. More especially, we discussed on the implementation details of the different configurations and their data structures. We also showed how to share code between configurations to save development efforts. Finally, we described an experimental protocol to compare fairly all configurations of the allDiffPrec constraint. We showed that, whenever domains can contain holes, our new filtering rule, in its bounds version (GodetBC) is the best compromise between execution speed and filtering strength. Unless domains are intervals, one should consider to use this implementation of the allDiffPrec constraint for greater performance. If the domains are intervals, experimental results showed that one should consider to implement the BessiereEtAl version of the allDiffPrec constraint.

Part III

Ordering variables to improve scheduling problem solving

Chapter 6

The generic ORDER constraint

In this chapter, we are going to discuss about a generic scheme on modelling and search whose objective is to greatly reduce the size of the search space: from pure exponential to factorial. Let us say that all variables have a domain of size d. Then the search space is of size $O(d^n)$. Depending on the value of d and the number of variables n, $n!$ might be way smaller than d^n, and as such the search space could be explored quicker. Let us remark that this observation can be mitigated depending on the filtering strength and the search heuristic.

6.1 Generic principle behind ordering variables

The scheme we describe in this chapter is based on what we call *list ordering algorithms*.

Definition 6.1 (List ordering algorithm)**.** *Given a problem P and a list L, a* list ordering algorithm *produces a solution of the problem from the list L.*

For instance, if we consider the one dimensional Bin Packing problem, famous algorithms as the First-Fit algorithm or the Best-Fit algorithm [Joh74] are both list ordering algorithms: from a list of items, they build a solution of the Bin-Packing problem, assigning items to bins such that no bin is overloaded.

A great source of list ordering algorithms can be found in approximation algorithms [Vaz03]. Approximation algorithms are algorithms that build a solution of a problem and such that the value of the objective-function of the built solution is guaranteed to be under a certain value: if we note $f()$ the objective-function to minimise, $f(S_{opt})$ the value of the objective-function on an optimal solution and S the built solution, then there exists $\alpha \in]1, \infty[$ such that $f(S) \leq \alpha f(S_{opt})$. These algorithms build a solution from the data of the problem, that can generally be expressed in the form of a list.

As the approximation algorithms consider that there exists an optimal solution that can be built by the approximation algorithm, then it is possible to build an optimal solution by looking for the list that would lead to an optimal solution when being given in parameter to the approximation algorithm. For convenience, we call such a list an *optimal list*. Whenever the size of the data of the problem is n, then the search space of all the possible lists is of size $n!$. With respect to the discussion on search space's size done in the introduction of the chapter, there are cases where looking for the optimal solution might lead to a quicker search than with the classical model of the problem.

In next section, we will discuss on a generic way on how to implement such an idea into constraint solvers.

6.2 The ORDER constraint: a constraint implementing list ordering reasoning

We consider a problem for which there exists a list ordering algorithm that can build an optimal solution. In the remaining of this chapter, we are going to present the ORDER constraint as well as a simple filtering algorithm. All we are going to present will stay generic, applications for several scheduling problems being presented in the next chapters.

We will note $X = \{x_1, \ldots, x_n\}$ the set of decision variables of the problem: these variables describe a solution of the problem when they are instantiated and all constraints are satisfied. Using the list ordering algorithm will instantiate these variables given some data and already instantiated variables.

The list representing the order into which variables would be given to the list ordering algorithm is modelled in the form of *permutation variables* $p_1, \ldots, p_n$, which set is noted P. Permutation variables take their values in $[1, n]$ such that $p_i = j$ means that variable x_j is at the i^{th} position within the list.

Definition 6.2. *Given a problem, we consider that there exists a function $f(X, L, i, j, Data)$ that returns the value assigned to x_j if placed at the i^{th} position in the list L given the $i-1$ preceding variables in the list L. Such a function is the base for the list ordering algorithm: given the list L, iteratively calling $f(X, L, i, L[i], Data)$ for $i \in [1, n]$ will build the solution. Data is a notation used for data specific to the problem, which should be specified for applications.*

The ORDER *constraint can be defined as:*

$$\text{ORDER}(X, P, f) \iff \forall i \in [1, n], x_{p_i} = f(X, P, i, p_i, Data) \tag{6.1}$$

The ORDER constraint represents as such the list ordering reasoning. Let us see how to filter inconsistent values for this constraint. Whenever permutation variables from p_1 to p_k are instantiated, the corresponding variables x_{p_1} to x_{p_k} should be instantiated according to the function f in order to respect the list ordering algorithm. If the corresponding variables cannot be instantiated to the value returned by function f, the propagator should fail during this phase.

Proposition 6.1. *The* ORDER *constraint respects the list ordering reasoning behind function f. As such, we have the following filtering rule:*

$$\forall i \in [1, n], \bigwedge_{1 \leq k \leq i} |D(p_k)| = 1 \implies D(x_{p_i}) \leftarrow D(x_{p_i}) \cap \{f(X, P, i, p_i, Data)\} \tag{6.2}$$

Proof. The proof is straightforward with the list ordering reasoning. □

Definition 6.3. *The* current permutation variable p_{idx} *is the permutation variable with the greatest index idx and such that all preceding permutation variables $p_1, \ldots, p_{idx-1}$ are instantiated and that is not instantiated itself: $idx = \min\{k \in [1, n] \mid |D(p_k)| > 1\}$.*

For every value j in the domain of the current permutation variable, if $f(X, P, idx, j, Data) \notin D(x_j)$, then x_j could not be placed at the current position without violating the ORDER constraint. From here, we deduce the filtering rule given in Proposition 6.2.

Proposition 6.2. *The* ORDER *constraint respects the list ordering reasoning behind function f. As such, we have the following filtering rule:*

$$\forall j \in D(p_{idx}), f(X, P, idx, j, Data) \notin D(x_j) \implies D(p_{idx}) \leftarrow D(p_{idx}) \setminus \{j\} \quad (6.3)$$

Proof. Let us suppose that j is not removed from $D(p_{idx})$ and such that $v = f(X, P, idx, j, Data) \notin D(x_j)$. If p_{idx} gets instantiated to j, then the filtering rule described in Equation 6.2 would remove all values in $D(x_j)$ except for v, which leads for $D(x_j)$ to be empty. □

From these two filtering rules, we can build a filtering algorithm, which is given in Algorithm 6.1.

Algorithm 6.1: Propagator for the ORDER constraint

```
1  Function Propagate(X : variables, P : permutation variables, f : function
     for the list ordering)
2  begin
3      i ← 1
4      do
5          while |D(p_i)| = 1 do
6              D(x_{p_i}) ← D(x_{p_i}) ∩ {f(X, P, i, p_i, Data)}
7              i ← i + 1
8          end
9          if i ≤ n then
10             for j ∈ D(p_i) do
11                 if f(X, P, i, j, Data) ∉ D(x_j) then
12                     D(p_i) ← D(p_i) \ {j}
13                 end
14             end
15         end
16     while |D(p_i)| = 1 ∧ i ≤ n
17 end
```

Line 9 tests if all permutation variables are instantiated or not. The algorithm, in order to be idempotent, as propagators in Choco solver should be, loops while the current permutation variable has been instantiated by the for loop from Line 10 to Line 14. For efficiency during the solving phase, the integer i should be made backtrackable in order to avoid running Line 5 to Line 8 each time the propagator is called. Instead, with i being backtrackable, these lines will be ran only when necessary, that is when the former current permutation variable has been instantiated through branching or by other constraints.

One can easily see that it is hard to use the list ordering reasoning to filter permutation variables succeeding the current permutation variable. As such, Algorithm 6.1 concentrates only on it, which can therefore not be characterised consistency-wise. Following the same idea, the search should focus only on the current permutation variable. As such, the inputOrder search is perfect for our case: it branches on the first uninstantiated variable in a list given in parameter. The given list is composed of the permutation variables P in the natural order. However, the value on which it is branched on should be computed depending on the problem and, possibly, on the

underlying idea of the list ordering algorithm.

Of course, one should note that the ORDER constraint is complementary with the classical model for the problem. As such, filtering from all constraints are used to reduce the search space. Whenever possible, one should also focus on potential additional constraints on permutation variables. Also, whenever possible, one should try to make the ORDER constraint exploit the filtering from other constraints to strengthen its own filtering: if the current state of domains of variables is not consistent with instantiations the ORDER constraint would make, filtering the corresponding variable from the current permutation variable should be done.

In following chapters, we will give several applications of this generic ORDER constraint, concentrating on the underlying list ordering reasoning as well as its implementation as a specialisation of the ORDER constraint.

Chapter 7

Application to the PMSPAUR

Initially, it was for this problem that the idea of looking for an optimal list ordering instead of the optimal schedule directly was thought out and introduced. It was only after that it was extended to the LEFTSHIFTED constraint (that will be presented in Section 8.1) and generalised into the ORDER constraint.

Interestingly, we can share that first implementations of the approach were made of two different propagators whose priorities were such that one propagator was executed before all the other ones and the second propagator was executed after all propagators have reached their fixpoint. The propagator that was executed first is the one that applies the filtering rule expressed in Proposition 6.1, while the one that was executed after all the others applies the filtering rule expressed in Proposition 6.2.

In this chapter, we will explore how these two propagators were thought out. In Section 7.1, we will give additional notations and notions that will be of use for our two new propagators. Section 7.2 presents a proof that a list ordering algorithm based on the `Enqueue` procedure [Heb+16] can yield an optimal schedule for the PMSPAUR. Then, we introduce some cutting rules in Section 7.3. In Section 7.4, we present how to implement the list ordering reasoning as well as the cutting rules in an efficient way. Experimental results are given in Section 7.5. Finally, we briefly conclude on these researches in Section 7.6.

The work presented in this chapter was published during the AAAI conference in 2020 [God+20]. We remind that the PMSPAUR was presented in Section 3.3.2.

7.1 Additional notations

In order to present the results on the PMSPAUR, we need additional notations, especially when it comes to prove the lemmas and theorems. The notions and notations presented here are complementary to the ones presented in Section 3.3.2.

A *schedule* is a mapping $\sigma : \mathcal{T} \mapsto \mathbb{N}$ from tasks to starting times. Let s_i be the starting time of task T_i, obviously we have $s_i = \sigma(i)$. σ is said feasible if and only if at any time $t \in \mathbb{N}$:

$$|\{T_i \mid s_i \leq t \leq e_i\}| \leq m \tag{7.1}$$

$$|\{T_i \mid T_i \in R_j, s_i \leq t \leq e_i\}| \leq 1 \; \forall j \in [1, s] \tag{7.2}$$

Equation (7.1) ensures that no more than m tasks can be simultaneously processed and equation (7.2) that no two different tasks requiring the same unit resource can

be simultaneously processed. Given a set of tasks $\mathcal{T}$ and a schedule σ, we denote[1] $e^{\mathcal{T}}_{min}$ the earliest idle time of any parallel machine, $e^{\mathcal{T}}_{max}$ the earliest time at which all parallel machines are idle, and $e^{\mathcal{T}}_{R_j}$ the latest usage time of resource R_j:

$$e^{\mathcal{T}}_{min} = \min\{t \mid t \in \mathbb{N} \ \wedge \ |\{i \mid s_i \leq t \leq e_i\}| < m\} \tag{7.3}$$
$$e^{\mathcal{T}}_{max} = \min\{t \mid t \in \mathbb{N} \ \wedge \ |\{i \mid s_i \leq t \leq e_i\}| = 0\} \tag{7.4}$$
$$e^{\mathcal{T}}_{R_j} = \max\{e_i \mid T_i \in R_j\} \tag{7.5}$$

Finally, we denote $L_{\mathcal{T}'}(R_j)$ the sum of the processing times of the tasks in $\mathcal{T}' \subseteq \mathcal{T}$ requiring resource j:

$$L_{\mathcal{T}'}(R_j) = \sum_{T_i \in R_j \cap \mathcal{T}'} d_i$$

7.2 Existence of a list ordering algorithm building optimal solutions

In this section, we recall some results from [Heb+16] and extend them in order to be used in a CP solver, most especially as an extension of the Order constraint. Algorithm 7.1 shows the basic procedure `Enqueue`.

Algorithm 7.1: `Enqueue` procedure

```
1 Function Enqueue(σ : schedule, T_i : task to insert)
2 begin
3 |   σ(i) ← max(e^T_min, e^T_res(T_i))
4 |   return σ
5 end
```

It simply inserts the task T_i given as argument at the "back" of the schedule, at the earliest possible time. More precisely, it schedules it at time t equals to the maximum between the earliest idle time of any machine $e^{\mathcal{T}}_{min}$ and the maximum usage time $e^{\mathcal{T}}_{res(T_i)}$ of $res(T_i)$. Applying the procedure on a sequence of the tasks gives a feasible schedule, as stated by Lemma 2 in [Heb+16]. Moreover, we get from Corollary 1 of the same paper that calling `Enqueue` on any sequence of tasks is a $(2 - \frac{1}{m})$-approximation algorithm, for m parallel machines.

In the remainder of this section, we prove that there exists a sequence of operations $\sigma \leftarrow$ `Enqueue`(σ, T_i) that yields an optimal schedule, and hence can be the basis for the use of the Order constraint. Moreover, we will show how to cut branches in the corresponding search tree.

Definition 7.1. *A schedule is* left-shifted *if no task can possibly be processed earlier without violating a resource constraint.*

Definition 7.2. *A schedule is* persistent *if every pair of tasks sharing a unit resource and immediately consecutive are processed on the same machine.*

Definition 7.3. *A schedule is* dense *if there is no $t_1 < t_2$ such that a machine is idle during $[t_1, t_2[$ and in process at t_2.*

[1] In the following, the schedule σ is always clear from the context, so we do not include it in the subscript.

Lemma 7.1. *There exists a left-shifted persistent dense optimal schedule.*

Proof. The fact that there exists an optimal left-shifted schedule is trivial.

Consider a left-shifted optimal schedule with two tasks T_i and T_j requiring the same unit resource such that $e_i = s_j$ but processed on two distinct machines, M_x and M_y, respectively. Then we can reassign T_j and all the trailing tasks on M_y to M_x, and all the tasks subsequent to T_i on M_x to M_y. Clearly, this operation changes no start time and cannot violate a resource constraint if it did not before the operation. The solution obtained is therefore equivalent and the operation can be repeated until there is no such occurrence. Therefore, there exists a left-shifted persistent optimal schedule.

Now, suppose that, in such a left-shifted persistent schedule, a machine M is idle in an interval $[t_1, t_2[$ and in process at time t_2. Since the task starting at time t_2 on this machine is left-shifted, then there must exist a task sharing the same unit resource and ending at time t_2 on some other machine. However, this would contradict the fact that this schedule is persistent. □

Theorem 7.1. *There exists a sequence of operations $\sigma \leftarrow$ `Enqueue`(σ, T_i) that leads to an optimal schedule.*

Proof. Let σ be a left-shifted persistent dense optimal schedule, order the tasks by their start times in σ in increasing order, and let us rename them accordingly.

We show by induction on the rank of the tasks in the ordering, that the corresponding sequence of calls to `Enqueue` produces σ (up to machine symmetries). This is trivially true for a single task. Suppose this is true until task T_i, and call σ_i the schedule restricted to tasks T_1 to T_i.

If the start time of T_{i+1} is the earliest idle time $t = e^{\mathcal{T}}_{min}$ of any machine in σ_i, then there is no task requiring $res(T_{i+1})$ in process after t. If there was such a task, since its start time is less than or equal to t, T_{i+1} could not start at t in σ. Therefore, `Enqueue` inserts that task on a machine with earliest idle time in σ_i yielding the same start time as in σ.

If the start time of T_{i+1} is not the earliest idle time $e^{\mathcal{T}}_{min}$ of any machine in σ_i, then, since σ is dense, no task $T_{i'}$, with $i' > i$, may be processed on the first-ending machine in σ. Therefore, since the schedule is left-shifted, all tasks subsequent to T_i must require one of the resources used at time s_{i+1} (of which there are at most $m - 1$). Since σ is persistent it has only one possible configuration: all the tasks of each resource are processed by a single machine. Since `Enqueue` does insert task T_{i+1} following the previous task requiring the same resource, task T_{i+1} will have the same starting time as in σ.

Therefore, there exists a sequence of calls to `Enqueue` that produces an optimal schedule. □

The search tree that explores the permutations of tasks to `Enqueue` is therefore guaranteed to contain an optimal solution.

Remark 7.1. *Moreover, from the proof of Theorem 7.1, we can observe that it is sufficient to explore only the orderings consistent with the chronological order in the resulting schedule. Notice that this might not be the case when the resource required by the next task T_i is in use at time t and there is another task $T_{i'}$ with $i' > i$ that might be inserted to start at time t. In that case, inserting first T_i and then $T_{i'}$ or the reverse yields the same schedule.*

7.3 Cutting rules for the PMSPAUR

In the subsequent paragraphs, we propose rules to reduce the size of the search space by cutting branches that are dominated. We assume that tasks are indexed by their order on the branch we consider. Therefore, when inserting T_i, previous tasks $T_1, \ldots, T_{i-1}$ are already allocated to machines and have a fixed start time, whereas following tasks are not scheduled yet.

Let us recall another algorithm presented in [Heb+16] that will be of use here: `MaxLoad` (see Algorithm 7.2).

Algorithm 7.2: `MaxLoad` algorithm

```
1  Function MaxLoad(𝒯: tasks)
2  begin
3      σ ← ∅
4      𝒯' ← 𝒯
5      while 𝒯' ≠ ∅ do
6          Let R_j be a resource with maximum load L_{𝒯'}(R_j)
7          Pick any task T_i ∈ 𝒯' ∩ R_j
8          𝒯' ← 𝒯' \ {T_i}
9          σ ← Enqueue(σ, T_i)
10     end
11     return σ
12 end
```

Theorem 7.2. *`MaxLoad` finds the optimal completion of any sequence of tasks if there are no more than m resources required by the remaining tasks.*

Proof. First, assume that the set of resources in use in the interval $[e^{\mathcal{T}}_{min}, e^{\mathcal{T}}_{max}[$ is disjoint from the resources required by the remaining tasks. In this case we can view the problem as a generalisation where each parallel machine M_z has a release date e_{M_z}.

We prove that `MaxLoad` is optimal in this case by induction on the number of machines m. For $m = 1$ this is trivial. Now suppose that this is true for m machines and consider the case for $m + 1$ machines. `MaxLoad` picks the resource R_j that maximises $L_{\mathcal{T}'}(R_j)$ and will eventually process every task requiring R_j on the machine M_x with earliest release time. If one of these tasks is the last completed task, then the solution is optimal, so we assume that the last completed task is on another machine M_y and requires R_k. The other choices made by the algorithm are completely independent of those on machine M_x and therefore we know that the other tasks are optimally scheduled on the m other machines. In particular, there is no resource except for R_j whose tasks we can swap for tasks requiring R_k in order to improve the schedule. However swapping the tasks of R_j with those of R_k does not help either since $L_{\mathcal{T}'}(R_j) \geq L_{\mathcal{T}'}(R_k)$ and the release date of M_y is larger than or equal to that of M_x.

Now, suppose that there exist resources in use during $[e^{\mathcal{T}}_{min}, e^{\mathcal{T}}_{max}[$ required by at least one remaining task. We can transform the instance as follows: for every machine M_z which is idle at time e_{M_z}, there is a unique resource in use in the interval $[e^{\mathcal{T}}_{min}, e^{\mathcal{T}}_{max}[$ (Lemma 2 in [Heb+16]). If this resource is required by some remaining task, we create a task of length $e_{M_z} - e^{\mathcal{T}}_{min}$ requiring that resource and we set the release time of this machine to $e^{\mathcal{T}}_{min}$. Otherwise, we simply set the release time of this machine to e_{M_z}. This new instance corresponds to the previous case and is thus

optimally solved by `MaxLoad`. Moreover, observe that the completion is such that every task requiring a given resource is processed by the same machine, and therefore the permutation of tasks on each machine does not matter. Therefore, this solution is also optimal for the original case. □

Let $L_{\mathcal{T}'}(\overline{R_j})$ denote the sum of the processing times of the tasks in $\mathcal{T}'$ that do not require resource R_j.

Lemma 7.2. *Let R_j be such that $L_{\mathcal{T}'}(R_j)$ is maximum. If $L_{\mathcal{T}'}(R_j) \geq 2L_{\mathcal{T}'}(\overline{R_j})/m$ then the minimum makespan of any schedule of $\mathcal{T}$ is $L_{\mathcal{T}'}(R_j)$.*

Proof. Trivially, $L_{\mathcal{T}'}(R_j)$ is a lower bound, therefore we only need to prove that the condition of the lemma entails that there is a solution with that makespan.

By Theorem 3 in [Heb+16] `MaxLoad` is a $(2 - \frac{2}{m+1})$-approximation algorithm. Moreover, the proof of this theorem shows that for any instance with set of tasks $\mathcal{T}'$, either:

1. one resource is in use at all times in the optimal schedule;
2. or the makespan of σ found by `MaxLoad` is at most $\left(2 - \frac{2}{m+1}\right) \frac{\sum_{T_i \in \mathcal{T}'} d_i}{m}$.

Now suppose that the condition of the lemma holds and consider the schedule where every task requiring resource R_j is processed on the same machine, and all the remaining tasks on the $m-1$ other machines. The makespan of this solution is therefore the maximum between $L_{\mathcal{T}'}(R_j)$ and the makespan C_{max} of the optimal solution of the sub-instance containing every task in $\mathcal{T}'$ that do not require R_j and only $m-1$ parallel machines. By Theorem 3 in [Heb+16], the optimal makespan C_{max} for this instance is either $L_{\mathcal{T}}(R_k)$ for some k, or at most $(2 - \frac{2}{m}) \frac{L_{\mathcal{T}'}(\overline{R_j})}{m-1} = \frac{2L_{\mathcal{T}'}(\overline{R_j})}{m}$. Since $L_{\mathcal{T}'}(R_j) \geq L_{\mathcal{T}}(R_k)$, we have in both cases $\max(L_{\mathcal{T}'}(R_j), C_{max}) = L_{\mathcal{T}'}(R_j)$. □

Lemma 7.3. *Let R_j be such that $L_{\mathcal{T}'}(R_j)$ is maximum and p_{max} denote the duration of the longest task not requiring R_j.*

If $L_{\mathcal{T}'}(R_j) \geq ((m-2)p_{max} + L_{\mathcal{T}'}(\overline{R_j}))/(m-1)$ then the minimum makespan is $L_{\mathcal{T}'}(R_j)$.

Proof. The proof is exactly the same as that of Lemma 7.2, except that it uses Theorem 2 instead of Theorem 3 in [Heb+16], where case 2 is: $C_{max} \leq (\sum_{T_i \in \mathcal{T}'} d_i)/m + \left(1 - \frac{1}{m}\right) p_{max}$ □

7.4 Implementation of the filtering and cutting rules in a new model for the PMSPAUR

The model we present in this section leverages the previous results in order to improve on the baseline constraint programming model. In particular, we add variables standing for the ordering of operations `Enqueue`, and constraints emulating their behaviour and channelling with the original variables. Backtrack search will then find a sequence of operations $\texttt{Enqueue}(\sigma, T_i)$ that yields an optimal schedule, the existence of which is assured by Theorem 7.1. To that extent, we use integer variables standing for a permutation: for all $k \in [1, n]$, p_k is an integer variable of domain $[1, n]$ such that $p_k = i$ indicates that task T_i is enqueued k^{th}. The set of permutation variables is noted P. The advanced CP model for the PMSPAUR is given in Figure 7.1.

$$\textit{minimise } C_{max} = \max_{i \in [1,n]} e_i \quad \textit{s.t.}$$

$$\forall k \in [1, s], \textsc{disjunctive}(\{T_i \mid res(T_i) = R_k\}) \tag{7.6}$$

$$\textsc{cumulative}(\mathcal{T}, m) \tag{7.7}$$

$$\textsc{allDifferent}(P) \tag{7.8}$$

$$\textsc{EnqueueCstr}(P, \mathcal{T}) \tag{7.9}$$

FIGURE 7.1: CP Model for the Parallel Machine Scheduling Problem with Additional Unit Resources (PMSPAUR) using permutation variables.

The ALLDIFFERENT constraint [Rég94] assures that P is a permutation of $[1, n]$. The ENQUEUECSTR constraint ensures the channelling between the permutation variables and the start variables. The dominance relation it enforces is correct by Theorem 7.1 and Remark 7.1.

7.4.1 ENQUEUECSTR

Definition 7.4. *The constraint* ENQUEUECSTR *ensures that start times are consistent with a sequence of* `Enqueue` *operations given by the permutation variables.*

$$\textsc{EnqueueCstr}(P, \mathcal{T}) \iff \forall i \in [1, n], s_{p_i} = \max\left(e_{\min}^{\{p_k | k<i\}}, e_{res(T_{p_i})}^{\{p_k | k<i\}}\right) \tag{7.10}$$

ENQUEUECSTR is therefore satisfied whenever the start time of each task corresponds to the value returned by `Enqueue` when calling it iteratively on the tasks following the permutation P. As such, ENQUEUECSTR can also be expressed as a specialisation of the ORDER constraint:

$$\textsc{EnqueueCstr}(P, \mathcal{T}) \iff \textsc{Order}(\mathcal{T}, P, \texttt{Enqueue}) \tag{7.11}$$

The propagation of this constraint is decomposed in two separate part: the *forward* channelling from the permutation variables to the start variables, and the *backward* channelling from start variables to permutation variables.

7.4.1.1 Forward Channelling

The forward channelling consists in instantiating the start variable $s_{p_{idx}}$ according to Definition 7.4 with p_{idx} being the current permutation variable as defined in Definition 6.3. More especially, the forward channelling corresponds to the filtering rule of Proposition 6.1 with `Enqueue` as the function f.

Proposition 7.1. *Forward channelling can be implemented to run in $O(n \times m)$ time over a branch.*

Proof. We can keep a backtrackable integer idx for the current permutation variable. When the variable p_{idx} is instantiated, we increment idx until $idx > n$ or p_{idx} is not instantiated, and we set the domain of every variable along the way to $\max(e^{\mathcal{T}}_{min}, e^{\mathcal{T}}_{res(T_{p_{idx}})})$.

The earliest idle time of each machine can be stored in a backtrackable integer t_x for each machine M_x. On top of that, two backtrackable integers store the earliest idle time $e^{\mathcal{T}}_{min}$ and a machine M_{idle} that is idle at $e^{\mathcal{T}}_{min}$. For each resource R_j, we maintain a backtrackable integer y_j that stores the machine on which resource R_j is the resource of the latest processed task. Whenever no task of R_j is scheduled, $y_j = -1$.

$e^{\mathcal{T}}_{res(T_{p_{idx}})}$ can be computed in $O(1)$, as it is stored in t_{y_j} if $y_j \neq -1$ or it is valued to 0 otherwise.

As $e^{\mathcal{T}}_{min}$ is stored in a backtrackable integer, it is therefore in $O(1)$ to know its value.

For forward channelling to be in $O(n \times m)$ over a branch, we need to prove that scheduling a task $T_{p_{idx}}$ with `Enqueue` can be done in $O(m)$. Whenever p_{idx} is instantiated, we determine the machine on which to schedule it in $O(1)$ as stated above (with $R_j = res(T_{p_{idx}})$, it is M_{idle} if $e^{\mathcal{T}}_{R_j} < e^{\mathcal{T}}_{min}$, or M_{y_j} otherwise). Let M_z be this machine. t_z is updated to $t_z + d_{p_{idx}}$, y_j to z and M_{idle} is updated in $O(m)$, which proves the proposition. □

Example 7.1. *Consider the instance described in Example 3.1.*

If we have $p_1 = 1$, $p_2 = 5$, $p_3 = 6$, $p_4 = 7$, *and* $p_5 = 8$, *the filtering algorithm of* EnqueueCstr *will instantiate* s_1, s_5, *and* s_6 *to* 0, s_7 *to* 4, *and finally* s_8 *to* 3. *The resulting partial schedule is shown in Figure 7.2.*

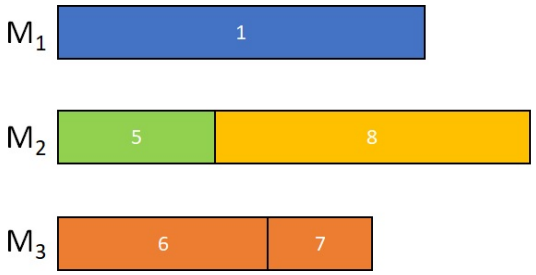


FIGURE 7.2: Partial schedule resulting from the permutation of Example 7.1

7.4.1.2 Backward Channelling

From Remark 7.1, we see that we can restrict search to sequences of `Enqueue` operations that are chronologically compatible. Therefore, if t is the minimum value in the domain of any task in $[1, n] \setminus \{p_k \mid k < idx\}$, then we can prune the value i from the domain of p_{idx} if the domain of s_i does not contain t. Indeed, the task whose start time can be t cannot require $res(T_i)$ and therefore, enqueueing this task or task T_i in any order yields the same schedule.

Proposition 7.2. *Supposing that* cumulative, disjunctive *and* allDifferent *constraints have reached their fixpoint, the filtering algorithm for backward channelling runs in* $O(n)$ *time.*

Proof. As CUMULATIVE and DISJUNCTIVE constraints have reached their fixpoint, the lower bound $\underline{s_i}$ of each start variable corresponds either to $e^{\mathcal{T}}_{min}$ or to $e^{\mathcal{T}}_{res(T_i)}$, as these constraints would have failed otherwise. Computing $t = \min_{i \in D(p_{idx})\}} \underline{s_i}$ is done in $O(n)$ as $|D(p_{idx})| \leq n$ (and values in $D(p_{idx})$ are all unscheduled tasks as the ALLDIFFERENT has reached its fixpoint). Removing all values $i \in D(p_{idx})$ such that $\underline{s_i} > t$ can also be done in $O(n)$. □

Remark 7.2. *Note that this dominance rule, and the associated filtering, must be called **after** that all other constraints have been propagated. This can be achieved by setting a high priority to the propagator, that is that the propagator is executed after all propagators for the* CUMULATIVE, DISJUNCTIVE *and* ALLDIFFERENT *constraints.*

Example 7.2. *Consider the same example as Example 7.1. The next permutation variable to be instantiated is p_6. In this example, t is equal to 7. Despite M_3 being idle at time 6, since there are tasks only from resources R_1 and R_4 that are not yet scheduled, the earliest time t at which a task can be scheduled is 7 for tasks of resource R_1 on machine M_1.*

Since $t = 7$, ENQUEUECSTR*'s backward filtering algorithm will remove values 9 and 10 from $D(p_6)$ because none of the corresponding start variables have 7 in their domains. Therefore, after the call to* ENQUEUECSTR*'s backward filtering algorithm, $D(p_6) = \{2, 3, 4\}$.*

7.4.2 Dominance rule using `MaxLoad`

As we have seen in Section 7.3, there are situations, described in Theorem 7.2, Lemma 7.2 and Lemma 7.3, where `MaxLoad` optimally completes the partial schedule. Therefore, we have a dominance rule on the permutation accepted by ENQUEUECSTR as in the stopping conditions described in Theorem 7.2, Lemma 7.2 and Lemma 7.3, the permutation is optimally completed by `MaxLoad`.

Proposition 7.3. *Let R_j be such that $L_{\mathcal{T}'}(R_j)$ is maximum, $L_{\mathcal{T}'}(\overline{R_j})$ denote the sum of the processing times of the tasks that do not require resource R_j and finally p_{max} the duration of the longest task not requiring R_j.*

*If $L_{\mathcal{T}'}(R_j) \geq \frac{2L_{\mathcal{T}'}(\overline{R_j})}{m}$ or if $L_{\mathcal{T}'}(R_j) \geq \frac{(m-2)p_{max}+L_{\mathcal{T}'}(\overline{R_j})}{m-1}$ or if there are no more than m resources required by the remaining tasks, then the current schedule can be completed by algorithm **`MaxLoad`** to the optimal schedule that we can get from the current state.*

Proof. This proposition is the direct application of Theorem 7.2, Lemma 7.2 and Lemma 7.3. □

Proposition 7.4. *The filtering rule described in Proposition 7.3 runs in $O(n \times (s + m))$, s being the number of unit resources and m the number of machines.*

Proof. Computing $L_{\mathcal{T}'}(R_k)$ for all resources R_k is done in $O(n)$. Determining R_j such that $L_{\mathcal{T}'}(R_j)$ is maximal as well as the value of $L_{\mathcal{T}'}(\overline{R_j})$ is done in $O(s)$. Computing p_{max} is done in $O(n)$. Looking if there are no more than m resources required by the remaining tasks is done in $O(s)$ (it is s minus the number of resources R_k for which $L_{\mathcal{T}'}(R_k) = 0$). Finally, completing the current schedule into an optimal one given the current state by algorithm `MaxLoad` is done in $O(n \times (s + m))$. Therefore the filtering rule described in Proposition 7.3 runs in $O(n \times (s + m))$. □

7.5 Experiments

This section is decomposed into three parts. First we introduce the benchmark configuration. Then, we present the six approaches that were evaluated. Finally, we report and discuss the results obtained by these approaches. The instances of the benchmark and the code developed for these researches can be found on GitHub [God19].

7.5.1 Experimental protocol

Our benchmark is composed of 234 instances of the Parallel Machine Scheduling Problem with Additional Unit Resources (PMSPAUR), all of which have been randomly generated. The randomness of the instances is on the processing time of the tasks, but is also on the number of tasks in each resource.

Number of machines m	Number of resources s
2	$3, 4$
3	$4, 5, 6$
5	$6, 7, 8, 10$
10	$12, 15, 17, 20$

TABLE 7.1: Configurations for m and s

Before generating the tasks and their processing times, we fixed several configurations for the number of machines and the number of resources, all of them are summarised in Table 7.1. The basic idea was to generate, for each number of machines, instances such that the number of resources was 1.25, 1.5, 1.75 and 2.0 times higher than the number of machines.

For each of these 13 configurations, we generated two types of instances: the first one has the same number of tasks in each unit resource (which is the case in our application, *i.e.* planning the download of acquisitions made by agile observation satellites); the second type has a random, positive, number of tasks in each unit resource.

We had two other ways to configure instances generation: the maximal number of tasks requiring a resource, and the maximal processing time of any task. The maximal number of tasks requiring a unit resource could be either 5, 10, or 20. The maximal processing time of a task could be either 10, 100, or 1000.

The generated instances have a total number of tasks between 6 and 400 and have very different shapes (size, number of machines and resources, short or large in time, etc.).

The classic model (Figure 3.3) is used in three configurations whose name comes from the search heuristic it is based on:: DOMOVERWDEG [Bou+04], SMALLEST and SETTIMES [Pap+94]. For remainder, *Smallest* selects the start variable with the smallest lower bound and branches on this particular variable, assigning it its lower bound. As *SetTimes* is close to the one we use in our approaches, we expect to find similar results in terms of efficiency. The main difference between the two approaches is that *SetTimes* will eventually branch on tasks that do not have minimum earliest start time (est). Say that you have two mutually exclusive tasks T_a and T_b, with est t and $t + 1$ respectively. *SetTimes* will schedule T_a first, and if failing, will not try to schedule T_a first again, but it will try to schedule T_b first and T_a second. In our model, T_a is the single candidate for the next insertion, and hence if that fails we will backtrack.

The Order and OrderA configurations are using the *inputOrderLB* search heuristic over permutation variables, which consists in selecting variables by order and selecting the lower bound of the selected variable as instantiation value (it is also called the lexicographic order branching scheme). These two configurations are based on the advanced model (Figure 7.1). Note that Order does not have the cutting rules described in Section 7.4.2 whereas OrderA does include them in the filtering of constraint EnqueueCstr.

The OrderAM model uses a custom search heuristic over permutation variables based on a Algorithm 7.2 to determine the value to assign to the selected variable. This configuration is also based on the advanced model (Figure 7.1).

All configurations were encoded and tested using *Choco Solver* [PFL17]. The model depicted in Figure 3.3 was also implemented on the *Chuffed lazy-clause generation solver* with its default search heuristic [Chu+16]. This last implementation is referred to as Chuffed in the results. The experiments were done on an Intel core i7-8650U (up to 4.2 GHz) processor. A time limit of 30 minutes was given for each configuration and each instance. The source code, the MiniZinc models and the data files at .dzn format are all available on the project repository [God19].

7.5.2 Experimental results

	domOverWDeg	Smallest	setTimes	Chuffed	Order	OrderA	OrderAM
#Proofs / #Solutions	86 / **234**	148 / **234**	127 / **234**	77 / 115	172 / **234**	174 / **234**	**213** / **234**
TimeToProof (ms)	48757	1933	1916	20329	1875	342	**137**
Objective	1.8780	1.0304	1.0617	1.5653	1.0201	1.0180	**1.0000**

Table 7.2: Results of the benchmark on the 234 generated instances

	domOverWDeg	Smallest	setTimes	Chuffed	Order	OrderA	OrderAM
#Proofs / #Solutions	56 / **86**	75 / **86**	81 / **86**	50 / **86**	81 / **86**	81 / **86**	**83** / **86**
TimeToProof (ms)	48910	765	828	19111	737	283	120
Objective	1.01638	1.00006	**1.00000**	1.01258	**1.00000**	**1.00000**	**1.00000**

Table 7.3: Results of the benchmark - small instances (86 instances of size < 40)

	domOverWDeg	Smallest	setTimes	Chuffed	Order	OrderA	OrderAM
#Proofs / #Solutions	25 / **97**	58 / **97**	39 / **97**	16 / 35	76 / **97**	76 / **97**	**87** / **97**
TimeToProof (ms)	45765	3926	3911	30463	3791	641	204
Objective	1.3960	1.0212	1.0624	1.4768	1.0016	1.0014	**1.0000**

Table 7.4: Results of the benchmark - medium instances (97 instances of size $40 \geq$ and < 120)

Table 7.2 reports an overall of the results obtained on the 234 generated instances. All our approaches obtain very good results at doing the proof of optimality for most of the instances. All configurations find at least one solution, whatever its quality, for every instance, except Chuffed that does not find any solution for 119 instances.

We can observe that our approaches are generally faster to prove optimality than any other approach. Moreover, in general, our approaches get better solutions, as shown by the line *Objective*. The values of the line *Objective* are computed as the

	domOverWDeg	Smallest	setTimes	Chuffed	Order	OrderA	OrderAM
#Proofs / #Solutions	5 / **51**	15 / **51**	7 / **51**	11 / 13	15 / **51**	17 / **51**	**43** / **51**
TimeToProof (ms)	59004	17490	15780	294	17166	231	199
Objective	4.2295	1.0975	1.1638	4.6521	1.0893	1.0801	**1.0000**

Table 7.5: Results of the benchmark - large instances (51 instances of size 120 $\geq$ and $\leq$ 400)

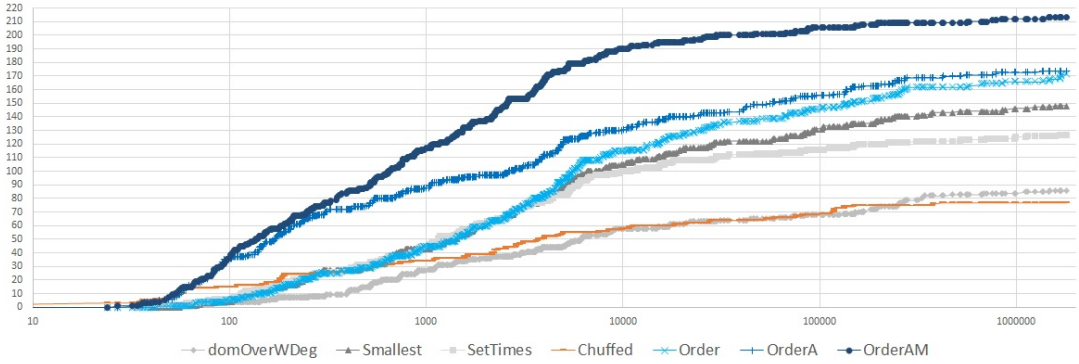


Figure 7.3: Number of proofs per milliseconds

mean of the ratio between the result of the approach (the makespan of the returned solution) and the best solution obtained among the seven configurations. We observe a clear advantage of our approaches (Order, OrderA and OrderAM), also with a clear improvement from Order to OrderA and from OrderA to OrderAM. It is interesting to note that the cutting rules allow to prove optimality at *root node* for 60 instances for OrderA and OrderAM. Analysing the results by considering the size of the problems in terms of tasks confirms the previous observations. Consequently, for respectively small instances (Table 7.3), medium instances (Table 7.4) and large instances (Table 7.5) observations remain the same.

Finally, we can remark from Table 7.2 that Order does almost as many proofs of optimality as OrderA, but Figure 7.3 shows that Order catches up with OrderA only near the time limit, which really shows the interest of the cutting rules described in Section 7.4.2. Of course, Table 7.2 and Figure 7.3 show the importance of a good search for a same model (OrderAM vs OrderA, or Smallest vs domOverWDeg).

7.6 Conclusion

In this chapter, we have proven that a sequence of `Enqueue` operations can yield an optimal schedule, which opens the way for an application of the Order constraint. Then, we gave three inequalities such that whenever one is true, the partial schedule can be optimally completed with the `MaxLoad` algorithm. Then, we showed how to implement the `Enqueue` procedure into a propagator as a specialisation of the Order constraint. We also discussed on the implementation of the cutting rules. Both these implementation are taking advantage of the list of priorities [SS08] implemented in Choco solver. We gave an extension of the CP model that uses these new constraints. Finally, we presented an experimental protocol and gave the results of different configurations of the state-of-the-art model and our new model. We analysed the results and showed that our new CP model gets better results than state-of-the-art configurations, therefore showing the interest of specifying the Order constraint. We also have

seen that dedicated cutting rules and heuristic search can greatly improve results, as the best results are obtained by the OrderAM configuration of our new model.

Chapter 8

Application to the RCPSP

The Resource-Constraint Project Scheduling Problem (RCPSP) is among the most known and studied scheduling problems. It is therefore normal to try our original approach on it. Of course, we were not expecting to get results as good as state-of-the-art methods, most especially LCG solvers using explanations of the CUMULATIVE constraint that close all classical instances from the literature [Sch+09; Sch+11; SFS13]. The idea of testing the specialisation of the ORDER constraint for the RCPSP was twofold. First, we wanted to see how well the idea thought for the PMSPAUR could adapt to another problem. On another hand, we wanted to compare the results of this approach to the classical model of the RCPSP as our method is quicker to implement than a complete framework to manage explanations. The specialisation of the ORDER constraint is therefore mostly addressed to CP solvers that do not embed explanations, or for instances with very large horizon, on which explanations might encounter difficulties because of memory consumption (this is true at least for the explanation of the decomposition of the CUMULATIVE constraint).

First, in Section 8.1, we will introduce the LEFTSHIFTED constraint as a specialisation of the ORDER constraint and explain how the filtering scheme works. Then we will show in Section 8.2 how to add this new constraint, as well as the list ordering reasoning from the ORDER constraint, to the CP model for the RCPSP. Experimental results are given in Section 8.3. Finally, we conclude on these researches in Section 8.4.

We remind that the RCPSP was presented in Section 3.3.1.

8.1 The LEFTSHIFTED constraint

8.1.1 Defining the LEFTSHIFTED constraint from the ORDER constraint

A common objective-function of scheduling problems is to minimize the makespan, *i.e.* the total time to process the schedule which is represented by the maximum of the end variables' value. Even for an optimal schedule with respect to the makespan, some tasks might have multiple available start time, even if all the other tasks stay fixed.

Definition 8.1 (Left-shifted)**.** *Let us consider a scheduled task T within a (partial) schedule. This task is* left-shifted *if it cannot be scheduled earlier without violating a constraint of the problem.*

A left-shifted schedule *is a schedule whose tasks are all left-shifted.*

The term left-shifted comes from the most common way to graphically represent a schedule: each task is a rectangle whose left side represents the start time, the right side the end time and the height of the rectangle corresponds to the task's

consumption of the given resource. As the time axis generally is represented from left to right, the notion of *left-shifted task* becomes quite obvious. The notion of left-shift was introduced in [SKD95], where the authors rather talk about *active schedule* than left-shifted schedule.

Example 8.1. *Consider a* CUMULATIVE *constraint whose maximal capacity is* 2*. In its scope, there are three tasks* T_1, T_2, T_3*.* T_1 *has a processing time of* 3 *(*$d_1 = 3$*) and* T_2 *and* T_3 *both have a processing time of* 2 *(*$d_2 = d_3 = 2$*). The tasks have the following resource consumption:* $h_1 = h_2 = h_3 = 1$*. This* CUMULATIVE *constraint is noted* c_1*.*

For simplicity, let us suppose that the only other constraint is the following precedence constraint: $e_1 \leq s_2$*. This precedence constraint is noted* c_2*.*

All tasks can be scheduled starting from time 0*. We consider a partial schedule such that* $s_1 = s_3 = 0$*. Both these tasks are left-shifted as they both start at time* 0*, the minimal possible value for all tasks, without violating* c_1 *nor* c_2*. According to the* CUMULATIVE *constraint* c_1*,* T_2 *can be scheduled from* $t = 2$ *as scheduling* T_2 *at time* 0 *or time* 1 *would lead to a momentary resource consumption of* 3 *where resource capacity is* 2*. However since* T_1 *ends at time* 3*, the precedence constraint* c_2 *indicates that* T_2 *cannot start before time* 3*.*

As such, T_2 *cannot start before time* 3 *without violating a constraint. Nothing prevents to schedule* T_2 *at a time* $t \geq 4$*. If* T_2 *is scheduled at time* 3*, then it is left-shifted and the schedule is also left-shifted. If* T_2 *is scheduled at a time* $t \geq 4$*, then it is not left-shifted as it could be scheduled earlier without violating any constraint.*

From the notion of left-shifted tasks and schedule, we can define a new constraint which has only left-shifted schedules for solutions. Definition 8.2 gives the formal definition of the LEFTSHIFTED constraint.

Definition 8.2. *Let* $\mathcal{T}$ *be a set of tasks variables and* C *the set of constraints that have these tasks variables in their scope. The* LEFTSHIFTED *constraint can be defined as:*

$$\text{LEFTSHIFTED}(\mathcal{T}, C) \iff \forall T_i \in \mathcal{T}, \forall t < s_i, \exists c \in C, \langle s_1, \ldots, s_{i-1}, t, s_{i+1}, \ldots, s_n \rangle \notin c \tag{8.1}$$

Basically, authorising only left-shifted schedules can lead to great cuts of the search tree. However, building a filtering scheme for the LEFTSHIFTED constraint can be difficult. Indeed, the LEFTSHIFTED constraint is a sort of *meta-constraint* as its definition is based on other constraints. We differentiate meta-constraints from constraints by the fact that a constraint has its own definition, whereas the definition of a meta-constraint depends on the definition of other constraints. As such, defining a filtering algorithm for a meta-constraint relies on filtering algorithms for underlying constraints. That said, meta-constraints can be considered as constraints as they behave similarly, having their own propagator.

Let us see how to build a filtering algorithm for the LEFTSHIFTED constraint as a derived constraint from the ORDER constraint presented in Chapter 6.

As it is already $\mathcal{NP}$-hard to find a solution of the CUMULATIVE constraint [GJ78; AB93], finding a support for the LEFTSHIFTED constraint cannot be done in polynomial time whenever one of the constraints is a CUMULATIVE constraint, which is very common in scheduling problems.

Definition 8.3. *Let us assume we have a partial schedule. Consider a constraint* c *and a time* t*. For a start variable* s *of a task* T*, we call the* minimum accepted value

$m_s^c(t)$ by the constraint c for the variable s the smallest time $t' \geq t$ such that assigning s at t' does not violate c.

We note $m_s^{C'}(t)$ the minimum accepted value by a set of constraints C' for a variable s, which is the smallest time $t' \geq t$ such that none of the constraints in C' is violated when assigning s to t'. Given a partial schedule, we note m_s the unique time at which s should be assigned to for its corresponding task to be left-shifted.

One can easily see the bridge with the ORDER constraint: m_s is the result of the problem-dependent function f on which relies the ORDER constraint. We will build the filtering algorithm for the LEFTSHIFTED constraint based on the notions of minimum accepted values.

Example 8.2. *We consider the case depicted in Example 8.1 with the partial schedule defined by $s_1 = s_3 = 0$. In this case, we have $m_{s_2}^{c_1}(0) = 2$ and $m_{s_2}^{c_2}(0) = 3$. From here, we want to recompute the minimum accepted values by each constraint to assure satisfaction. We have $m_{s_2}^{c_1}(3) = 3$ and $m_{s_2}^{c_2}(3) = 3$, and therefore $m_{s_2} = 3$.*

One way to build a (partial) left-shifted schedule is to consider a list L of the tasks. Following this list, tasks are iteratively made left-shifted by computing the minimum accepted value from the partial left-shifted schedule composed of the already scheduled tasks. This constitutes our list ordering algorithm, the function f here returning m_s for a given start variable s. This list ordering algorithm is named `Serial Scheduling Scheme (SSS)` was introduced by Kelley in [Jr63]. Kolisch showed in [Kol96] that it produces left-shifted schedule for the RCPSP.

Therefore, we can build filtering algorithms for the LEFTSHIFTED constraint based on the ones we described in Proposition 6.1 and Proposition 6.2, and encoded in Algorithm 6.1, as the LEFTSHIFTED constraint can be seen as a specialisation of the ORDER constraint using the `Serial Scheduling Scheme`:

$$\text{LEFTSHIFTED}(\mathcal{T}, C) \iff \text{ORDER}(\mathcal{T}, P, \texttt{SSS}) \tag{8.2}$$

8.1.2 Implementation for precedence constraints

Given a partial schedule and a task variable T_j, it is not hard to compute $m_{s_j}^c(t)$ for a precedence constraint c. One should simply loop over the predecessors of the task T_j. $m_{s_j}^c(t)$ is the maximum value among all predecessors' end time and time t: $\forall i \in [1, n], \forall j \in D(p_i), m_{s_j}^{(k,j)\in\Gamma^-}(t) = \max(t, \max_{k \in \{p_1, \ldots, p_{i-1}\}} e_k)$.

8.1.3 Implementation for CUMULATIVE constraints

For the CUMULATIVE constraint, it is essential for solver performance to build the partial schedule when looping over positioned tasks indicated by the instantiated permutation variables. Of course, if integer i has been made backtrackable, the partial schedule should also be backtrackable.

To compute $m_{s_j}^c(t)$ for a given CUMULATIVE constraint c, we look for the smallest value $t' \geq t$ such that scheduling task T_j at time t' does not violate the CUMULATIVE constraint's capacity. In our case, we relied on the Profile data structure presented in [GHS15] which itself is built using a sweep-line algorithm [BC02; LBC12]. We loop over the timestamps from t' at which there is a resource consumption change (start of a scheduled task or end of a scheduled task) until we found one at which scheduling task T_j would not violate the resource's capacity.

8.1.4 Complexity to compute the minimum accepted value for a start variable for precedence and CUMULATIVE constraints

Note that computing the minimum accepted value for task T_j over precedence constraints can be done once and for all: it is $\max_{\substack{(k,j)\in\Gamma^- \\ k\in\{p_1,\ldots,p_{i-1}\}}} e_k$. If CUMULATIVE constraints cause a delay, then the new value will prevail over what is computed for precedence constraints. One should therefore compute the minimum accepted value for task T_j over precedence constraints once and for all and then loops only on the CUMULATIVE constraints. As T_j has at most $n-1$ predecessors, computing m_{s_j} over precedence constraints can be done in $O(n)$.

It is possible to find the m_{s_j} over CUMULATIVE constraints without looping over the different CUMULATIVE constraints while one constraint delays the minimum accepted value for s_j. Let's build a Profile data structure for all the CUMULATIVE constraints at the same time: each rectangle indicating the consumption over the different resources for its duration. Therefore, looping over the rectangles, we can check all the CUMULATIVE constraints.

To check if a task T_j can be scheduled at the beginning of a rectangle, we look at the resources' availability from the different resources. If it exceeds the capacity of one resource, then the next rectangle is checked. If it can be scheduled at the time corresponding to the start point of the rectangle, we study two cases: either the task is longer than the rectangle's duration, either not. In the case where the task has a smaller processing time than the rectangle's duration, then the task can be scheduled at the rectangle's start. Whenever the task's processing time is greater than the rectangle's duration, then we consider a new artificial task T' whose processing time is the processing time of task T_j minus the duration of the rectangle, and we do the same checking process with the artificial task T' on the next rectangle. As such, checking if a task T_j can be scheduled at the beginning of a rectangle can be done in $O(k \times d_j)$ with k the number of CUMULATIVE constraints.

As there are at most $O(n)$ rectangles in the Profile data structure, the worst-case time complexity to compute m_{s_j} for precedence and CUMULATIVE constraints is in $O(n \times k \times d_j)$ for a task T_j.

8.2 Using the LEFTSHIFTED constraint to solve the RCPSP

Here, we build over the classic model for the RCPSP, presented in Figure 3.1, using the LEFTSHIFTED constraint to reduce the size of the search space, hoping therefore for better solving performance.

Lemma 8.1 ([Kol96]). *There exists an optimal left-shifted schedule for the RCPSP.*

The Lemma 8.1 shows a way to improve the classic model. Any left-shifted solution of the RCPSP can be represented as a permutation of the activities. Indeed, building the schedule can then be done by left-shifting the activities following the order given by the permutation. Therefore looking only for left-shifted schedule is equivalent to looking for one optimal permutation of the activities. To this end, we introduce *permutation variables* P, each of domain $[1, n]$, as expected for the filtering algorithm for the ORDER constraint (Algorithm 6.1).

And, of course, the LEFTSHIFTED constraint is also added to the model. As explained for implementation of the LEFTSHIFTED constraint in Section 8.1, given the constraints applied to task variables, different implementation, and therefore data, are needed. Here, we indicate all needed data and variables for the LEFTSHIFTED

constraint to work for the RCPSP. The permutation variables are used to represent the partial schedule as explained earlier, the set of tasks $\mathcal{T}$ give of course information linked to start and end variables, the set of consumption variables and the set of resources' capacities are used to build the partial schedule used for the Profile data structure, and finally the set of precedence Γ^- is used to compute the minimum accepted value by precedence constraints for each start variable.

The permutation variables entirely describe solutions, as such the search space of this model is of size $O(n!)$.

Consider a simple example. There are 4 tasks $T_1, \ldots, T_4$, T_1 precedes T_2 and no other precedence are considered. All tasks have a duration of 2. We consider two resources R_1 and R_2, such that $h_{1,1} = h_{2,2} = h_{3,2} = h_{4,1} = 1$ and all other consumption variables are equals to 0. The permutation $[1, 2, 3, 4]$ leads to start values $s_1 = s_3 = 0, s_2 = s_4 = 2$, which leads to an optimal makespan of 4. The permutation $[1, 3, 2, 4]$ also leads to a makespan of 4 but tasks are sorted by increasing start time within the permutation. In order to additionally improve the model, we consider the following symmetry-breaking rule: tasks should be sorted by increasing start time within the permutation.

To implement the symmetry-breaking rule, we introduce *precedence variables* $\hat{O}$, which are boolean variables, $\hat{o}_{i,j}$ being `true` whenever activity a_i starts before activity a_j, $\hat{o}_{i,j}$ being `false` otherwise. The permutation respects the precedence property: an activity a_i cannot appear before its predecessors within the permutation, as it would otherwise start before one of its predecessors and therefore violate a precedence constraint. Reflecting such precedence on the permutation variables can be done through a GeneralizedAllDiffPrec constraint, as presented in Chapter 4, and an inverse constraint, the latter one being defined as: $\text{inverse}(\{x_1, \ldots, x_n\}, \{y_1, \ldots, y_n\}) \iff \forall i, j \in [1, n], x_i = j \iff y_j = i$. Doing so needs to introduce intermediary variables, which we note $\{z_1, \ldots, z_n\}$ here.

The Figure 8.1 shows the complete CP model using permutation variables for the RCPSP.

8.3 Experiments

8.3.1 Experimental protocol

For each instance, we compare the performance between the two models described in Figure 3.1 and Figure 8.1. For the classic model described in Figure 3.1, we use two different search heuristics: *Smallest* and *SetTimes* [Pap+94].These configurations will be referred to as Smallest and setTimes. For the model described in Figure 8.1, we use a decomposition of the GeneralizedAllDiffPrec constraint into an allDifferent constraint and precedence constraints, all enforcing bounds(Z) consistency. The search we use branches on the current permutation variable p_{idx}. The selected value to branch on corresponds to the start variable with the smallest lower bound: $v \in D(p_{idx})$ such that $\underline{s_v} = \min(\{\underline{s_k} \mid k \in D(p_{idx})\})$. This configuration will be referred to as Decomposition. When the GeneralizedAllDiffPrec is implemented using the fixed version of the algorithm of Bessiere et al. (Algorithm 4.5) as well as an allDifferent and precedence constraints enforcing bounds(Z) consistency, the configuration is noted BessiereEtAl. Finally, when the GeneralizedAllDiffPrec

$$minimise\ C_{max} = \max_{i \in [1,n]} e_i \quad s.t.$$

$$\forall (i,j) \in \Gamma^-, e_i \leq s_j \tag{8.3}$$

$$\forall k \in [1;r], \textsc{cumulative}(\{T_i \mid R_k \in res(T_i)\}, \{h_{i,k} \mid R_k \in res(T_i)\}, C_k) \tag{8.4}$$

$$\textsc{GeneralizedAllDiffPrec}(\{z_1, \ldots, z_n\}, \hat{O}) \tag{8.5}$$

$$\textsc{inverse}(P, \{z_1, \ldots, z_n\}) \tag{8.6}$$

$$\forall 1 \leq v < w \leq n, \hat{o}_{v,w} = 1 \iff s_v \leq s_w \tag{8.7}$$

$$\textsc{leftShifted}(P, \mathcal{T}, \{h_{i,k} \mid T_i \in \mathcal{T}, R_k \in \mathcal{R}\}, \{C_k \mid k \in [1;r]\}, \Gamma^-) \tag{8.8}$$

FIGURE 8.1: CP Model for the Resource-Constrained Project Scheduling Problem (RCPSP) using permutation variables.

constraint is implemented using the BC version of the approach presented in Section 4.5 of Chapter 4 on the GeneralizedAllDiffPrec constraint, we note this configuration GodetBC.

The instances we work on for the RCPSP are the classic ones from literature. Despite they come from other papers, all the instances are centralised on one database: the PSPLib [KS97]. We concentrated only on "hard" instances, *i.e.* the instances for which at least the configuration Smallest or the configuration setTimes could not do the proof of optimality within 5 minutes. Our RCPSP benchmark was so composed of 126 $j30$ instances (out of 480), 143 $j60$ instances (out of 480), 144 $j90$ instances (out of 480) and 448 $j120$ instances (out of 600). The detailed list of instances that were used can be found in the GitHub repository [God21a].

Initially, we wanted to include the *Failure-Directed Search* of Vilim et al. [VLS15] in our benchmark. However, as it is a "plan B" strategy, which was experimentally confirmed on many instances, the statistics would be mostly incomparable. It is interesting to note that the model described in Figure 8.1 tends to give very good solutions, and as so can be used as the "plan A" strategy that would be completed by *Failure-Directed Search*.

The Choco-solver library [PFL17] was used in its 4.10.5 version. A time limit of 30 minutes was given for each configuration and each instance.

Every instance and piece of code that were used can be found in the GitHub repository [God21a].

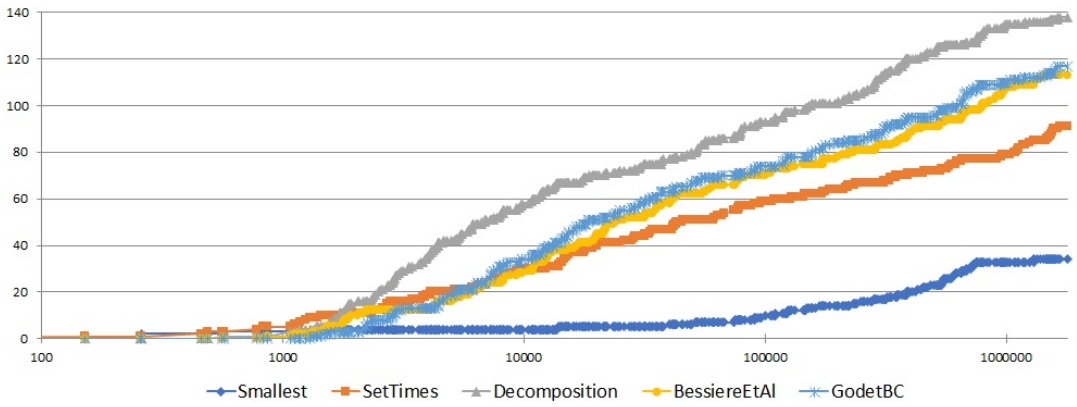


FIGURE 8.2: Number of proofs per milliseconds on the RCPSP instances

8.3.2 Experimental results

The results are shown in the form of a graphic of the number of proofs per milliseconds as well as tables that give more precise data. The first line of the table gives the number of proofs that were done in the time limit of 30 minutes by each configuration. The second line gives the mean time to do the proof when all approaches did the proof (SMALLEST is excluded as it does too few proofs to allow a fair comparison between the other configurations). The third line gives the mean ratio of the best makespan found by the configuration on the best makespan found among the configurations. For instance, if we note $C^c_{max}(i)$ the best makespan found by configuration c in the time limit for instance i, then the ratio $r_c(i)$ of configuration c for instance i would be: $r_c(i) = \frac{C^c_{max}(i)}{\min_{c' \in \mathcal{C}}(C^{c'}_{max}(i))}$ with $\mathcal{C}$ being the set of configurations. The line Objective gives the mean ratio of each configuration across all instances. Finally, the two last lines give the number of instances for which the configuration finds (strictly) the best found makespan among all configurations. All results are given in Figure 8.2, Table 8.1, Table 8.2, Table 8.3, Table 8.4 and Table 8.5.

	SMALLEST	SETTIMES	DECOMPOSITION	BESSIEREETAL	GODETBC
#Proofs	34	91	**138**	113	117
TimeToProof (ms)	/	248241	**24972**	112613	81867
Objective	1.0198	1.0304	**1.0029**	1.0104	1.0095
Nb times strictly best	0	94	**292**	0	0
Nb times best	287	289	**767**	407	438

TABLE 8.1: Results of the benchmark on the 861 RCPSP instances

	SMALLEST	SETTIMES	DECOMPOSITION	BESSIEREETAL	GODETBC
#Proofs	20	65	**90**	82	84
Objective	1.0363	1.0058	**1.002**	1.0072	1.0067
Nb times strictly best	0	8	**9**	0	0
Nb times best	55	104	**118**	102	103

TABLE 8.2: Results of the benchmark on the 126 j30 RCPSP instances

	Smallest	setTimes	Decomposition	BessiereEtAl	GodetBC
#Proofs	4	13	**22**	18	19
Objective	1.0311	1.0228	**1.0069**	1.0172	1.0157
Nb times strictly best	0	29	**35**	0	0
Nb times best	35	66	**114**	64	71

Table 8.3: Results of the benchmark on the 143 j60 RCPSP instances

	Smallest	setTimes	Decomposition	BessiereEtAl	GodetBC
#Proofs	4	5	**10**	7	8
Objective	1.0183	1.0297	**1.0029**	1.0107	1.0099
Nb times strictly best	0	14	**56**	0	0
Nb times best	43	40	**130**	63	69

Table 8.4: Results of the benchmark on the 144 j90 RCPSP instances

	Smallest	setTimes	Decomposition	BessiereEtAl	GodetBC
#Proofs	6	8	**16**	6	6
Objective	1.0121	1.0399	**1.0019**	1.0091	1.0082
Nb times strictly best	0	43	**192**	0	0
Nb times best	154	79	**405**	178	195

Table 8.5: Results of the benchmark on the 448 j120 RCPSP instances

Globally, we can see from the different tables that the search space reduction allowed by the model described in Figure 8.1 offers really good solving performance compared to the classical model described in Figure 3.1. Indeed, all configurations based on the model described in Figure 8.1 do a greater number of proofs than Smallest and setTimes. Moreover, whenever setTimes, Decomposition, BessiereEtAl and GodetBC all prove optimality of their best found solution, we can see that all configurations based on Figure 8.1 are at least twice faster than setTimes, Decomposition being the faster configuration overall and being 10 times faster as setTimes.

Interestingly, when looking at the results instance by instance, we can see that Decomposition is always better than GodetBC, which is in turn always better than BessiereEtAl. This can be explained by the fact that the better filtering offered by BessiereEtAl and GodetBC on $\{z_1, \ldots, z_n\}$ variables do not lead to significant cut in the search space. In fact, despite doing some filtering sooner in the search tree, BessiereEtAl and GodetBC explore, for most instances, as many nodes as Decomposition to find the best solution and to do the proof. The difference of performance is thus explained by the great practical speed of the decomposition of the allDiffPrec constraint, while the greater filtering of GodetBC seems to offer a greater trade-off than the greater speed of BessiereEtAl, whenever greater filtering influences the exploration of the search tree. In our current use-case, it seems that the decomposition of the GeneralizedAllDiffPrec constraint offers the best trade-off between speed and filtering.

An important remark to do is that the great results presented here rely heavily on the symmetry breaking over precedence variables $\hat{O}$ and the use of the

GeneralizedAllDiffPrec constraint instead of a simple allDiffPrec constraint. Indeed, using fixed precedence boolean O and an allDiffPrec constraint led to more mitigated results, when not worse. This can be easily understood by the fact that branching on permutation variables without precedence variable might not lead to greater filtering than the classic model and might as such lead to explore a greater number of nodes.

8.4 Conclusion

In this chapter, we introduced the leftShifted constraint, and showed how to implement it as a specialisation of the Order constraint. Most especially, we gave some insights on the working of a filtering scheme for the leftShifted constraint when it is based on precedence and cumulative constraints, as it is the case for the RCPSP. We also discussed on the complexity of this filtering scheme. Then, we gave a CP model for the RCPSP which is the classical model described in Figure 3.1 and that have additional constraints and variables. Indeed, we introduced the permutation variables, on which precedence constraints can also be applied and is therefore an application case of the GeneralizedAllDiffPrec constraint introduced in Chapter 4. Of course, the leftShifted constraint introduced earlier is also part of these additional constraints. Finally, we presented an experimental protocol and gave the results of different configurations of the state-of-the-art model and our new model. Most especially, we saw that, as expected from the discussions and analyses of Chapter 5, the decomposition of the GeneralizedAllDiffPrec would most benefit here as the intermediary variables should not contain holes in their domains. Our model obtained good results compared to the state-of-the-art one, but it is more mitigated than on the PMSPAUR as some instances are still better solved by the state-of-the-art model. All in all, our model is the best one tested in average and is a good application case of the Order constraint and, more generally, application case of list ordering reasoning in CP to solve scheduling problems.

Note that the dominance rule of accepting only left-shifted schedules was already explored by Chu and Stuckey [CS15] for the RCPSP. They elegantly modelled the dominance rule with an element constraint [HC88]: any start variable s_i takes its value in an array composed of a variable instantiated to 0 (or the origin of time of the problem more generally) and all the end variables $e_1, \ldots, e_n$. Indeed, in the case of the RCPSP, any left-shifted task T_i starts either at the beginning, either whenever another task finishes, whether because it is one of its predecessors or because it thus liberates enough resource for the task T_i to start. The existence of this particular paper was pointed to us late in our research, which is why we did not compare to this implementation of the leftShifted constraint. However, we do not have any doubt that our implementation, as a specialisation of the Order constraint, would probably face difficulty when compared to the standard model improved with an element constraint as described above. Further research can validate such thoughts. Nevertheless, this chapter still stands as it shows another example of specialisation of the Order constraint.

Chapter 9

Application to the UET-UCT

The Unit Execution Time-Unit Communication Time (UET-UCT) is a very specific problem. As discussed in Section 3.3.3, it is a simplified version of the more general problem of scheduling tasks on identical machines. The reason why we chose the UET-UCT instead of the more general problem is that we were expecting the experimental results to be far more mitigated for the UET-UCT. Indeed, the fact that both the execution and communication times are unitary implies that the horizon is shorter and therefore schedules tend to be tight for the classical model and having more impact on the domains' filtering. The idea was thus to greatly stress the Order constraint's specialisation and see its limits.

First, in Section 9.1, we will show how to improve the filtering done in the propagation phase using some additional constraints. The idea is to make better inference on the assignment variables, as they are deeply involved in the combinatorial of the problem. Then we will show in Section 9.2 how to adapt the List Algorithm [Pin12] to the duplication UET-UCT using the notion of D-path of Munier and Hanen [MH97]. We thereafter show how to introduce the resulting list ordering algorithm into a propagator, therefore specifying the Order constraint for the duplication UET-UCT. We will also show how we can take advantage of the filtering of other constraints in the filtering algorithm of our new propagator. Experimental results are given in Section 8.3. Finally, we conclude on these researches in Section 8.4.

We remind that the UET-UCT was presented in Section 3.3.3.

9.1 Additional constraints to help filtering on assignment variables

First, note that the inference of constraints on the model described in Figure 3.4 misses some easy-to-catch information. Let us see it in an example.

Example 9.1. *Consider the following example of UET-UCT with duplication for* 8 *unit tasks. We consider the precedence described in Figure 9.1. The number of machines is* $m = 3$*. Finally, we consider the case where a solution with a makespan of* 5 *has already been found and we are therefore looking for a solution with a makespan of* 4*, should one exist. The domains of the start variables are as following:* $D(s_0) = \{0\}$*,* $D(s_1) = \{0, 1, 2, 3\}$*,* $D(s_2) = \{0, 1, 2, 3\}$*,* $D(s_3) = \{1\}$*,* $D(s_4) = \{1\}$*,* $D(s_5) = \{2\}$*,* $D(s_6) = \{3\}$ *and* $D(s_7) = \{3\}$*. There are still the values assigned to* s_1*,* s_2 *and assignment variables* $b_{0,1}, \ldots, b_{7,3}$ *to determine. Consider the case where* $b_{0,1} = 1$*,* $b_{0,2} = 1$*,* $b_{3,1} = 1$*,* $b_{4,2} = 1$ *and* $b_{5,1} = 1$*. All other assignment variables are undefined.*

Through propagation, the solver induces that $b_{3,2} = 0$*,* $b_{4,1} = 0$*,* $b_{5,2} = 0$*,* $b_{6,2} = 0$ *and* $b_{7,2} = 0$*. The rest of the assignment variables are still not instantiated. However,*

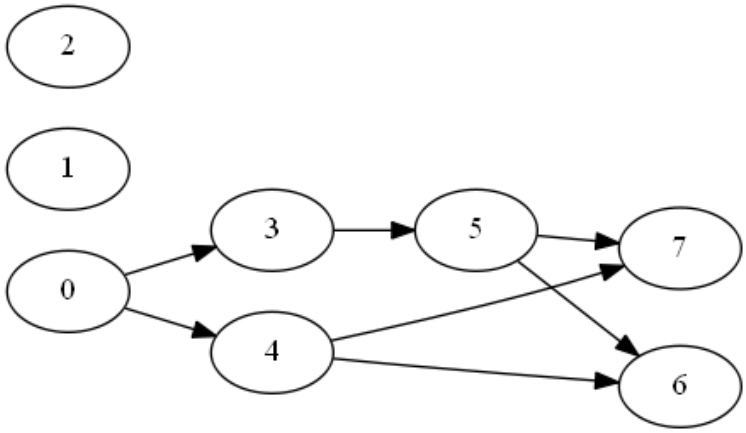


FIGURE 9.1: Precedence graph of an UET-UCT instance

from the values of the variables s_6 and s_7, we can deduce that their common predecessor T_5, which is processed only one time unit before T_6 and T_7, should be executed on two machines for T_6 and T_7 to be executed at time 3. As $b_{5,2} = 0$, we know that $b_{5,3} = 1$, which would thus enforce $b_{0,3} = 1$ and $b_{3,3} = 1$. Depending on the filtering rules of the DISJUNCTIVE *constraint, the propagation phase could also determine from here that $s_1 \geq 2$ and $s_2 \geq 2$, as well as $b_{1,1} = 0$, $b_{1,3} = 0$, $b_{2,1} = 0$, $b_{2,3} = 0$, which would lead to $b_{1,2} = 1$ and $b_{2,2} = 1$ with Equation 3.6.*

Example 9.1 shows that additional constraints can greatly improve filtering. Therefore, from here, we consider that the CP model described in Figure 3.4 also has the following constraints:

$$\forall i \in [1, n], \sum_{k=1}^{m} b_{i,k} \geq |\{j \in \Gamma^+(i) \mid s_j = s_i + 1\}| \tag{9.1}$$

Indeed, these constraints enforce that there are at least as many copies of task T_i that there are tasks T_j that directly follow T_i in the schedule.

Let us also note that some lower bounds on the makespan can easily be applied at root node. Indeed, the makespan is at least the number of tasks divided by the number of machines and, as we consider unit execution time tasks, is at least as high as the longest chain in the precedence graph, which we note d:

$$C_{max} \geq \max(\left\lceil \frac{n}{m} \right\rceil, d) \tag{9.2}$$

From now on, we will also consider that these lower bounds have been applied on the makespan. All in all, it leads to the CP model described in Figure 9.2.

9.2 Applying ORDER to the UET-UCT

The idea to build a schedule from an ordered list of the tasks for the UET-UCT is not new. Indeed, Pinedo already presented what he called a *List Algorithm* [Pin12], which builds a schedule for the non-duplication UET-UCT from an ordered list of the tasks. Zinder et al. [Zin+10] built on this idea and proved that there exists an *optimal list* such that an optimal schedule is built by the List Algorithm. They also gave a branch-and-bound procedure to find such an optimal list. Sadly, their procedure cannot be generalised to the duplication UET-UCT as they use specific

$$\textit{minimise } C_{max} = \max_{i \in [1,n]} e_i \quad \textit{s.t.}$$

$$\forall i \in [1,n], s_i + 1 = e_i \tag{9.3}$$

$$\forall i \in [1,n], \sum_{k=1}^{m} b_{i,k} \geq 1 \tag{9.4}$$

$$\forall k \in [1,m], \text{DISJUNCTIVE}(\mathcal{T}, \{b_{1,k}, \ldots, b_{n,k}\}) \tag{9.5}$$

$$\forall i \in [1,n], \forall j \in \Gamma^+(i), s_i + 1 \leq s_j \tag{9.6}$$

$$\forall i \in [1,n], \forall j \in \Gamma^+(i), s_i + 1 = s_j \implies \forall k \in [1,m], b_{i,k} \geq b_{j,k} \tag{9.7}$$

$$\forall i \in [1,n], \sum_{k=1}^{m} b_{i,k} \geq |\{j \in \Gamma^+(i) \mid s_j = s_i + 1\}| \tag{9.8}$$

$$C_{max} \geq \max(\left\lceil \frac{n}{m} \right\rceil, d) \tag{9.9}$$

FIGURE 9.2: CP Model for the Unit Execution Time-Unit Communication Time (UET-UCT) with additional constraints to improved filtering

lower bounds computations. Note that their branch-and-bound procedure looks like more of a meta-heuristic than a complete exploration procedure usually behind the branch-and-bound name.

In the remaining of this section, we will see how to build a list ordering algorithm for the duplication UET-UCT by adapting an approximation algorithm from Munier and Hanen [MH97]. This list ordering algorithm will then be used within a specialisation of the ORDER constraint for the duplication UET-UCT.

9.2.1 D-path and list ordering algorithm for the duplication UET-UCT

To face the possibility of duplicating tasks, Munier and Hanen [MH97] introduced the notion of *D-path*.

Definition 9.1 (D-path [MH97])**.** *We consider a partial schedule σ and a task T_i such that all its predecessors are scheduled into σ. We note t_i the greatest time at which a predecessor of T_i has been scheduled: $t_i = \max_{j \in \Gamma^-(i)} s_j$. The* D-path *$D_i$ of T_i is a subset of the precedence graph. If there are two tasks or more among T_i's predecessors that*

are scheduled at t_i, then the D-path *D_i of T_i is $\emptyset$. If there are at most one predecessor T_j of T_i that is scheduled at t_i, the* D-path *D_i of T_i is the longest subpath of the precedence graph that ends by T_j and such that for any arc $(k, k') \in D_i$, $s_{k'} = s_k + 1$. The first task in D_i is noted l_i and is clearly such that $\forall k \in \Gamma^-(l_i), s_{l_i} - s_k \geq 2$.*

Any task T_k that is part of the D-path D_i has at most one predecessor scheduled at $s_k - 1$, which makes the D-path D_i unique for all $i \in [1, n]$.

Let us remind some properties of the D-path that Munier and Hanen gave in [MH97].

Proposition 9.1 (Properties of the D-path [MH97])**.** *Let us consider a partial schedule and a task T_i all predecessors of which have been scheduled and such that its D-path D_i is not empty. We still note t_i the greatest time at which a predecessor of T_i is scheduled. We have the following properties:*

- *Let π be a machine which is free during the interval $[t_i+1, t_i+2)$. T_i is schedulable at time $t_i + 1$ on π if and only if every task of D_i is scheduled on π.*
- *Let $k \in D_i$ be a task scheduled on π. If π is free during the interval $[t_k+1, t_i+2)$, then T_i can be scheduled on π at time $t_i + 1$ by duplicating on π successors of k in D_i. D_i is then entirely performed on π.*
- *If π is free during the interval $[t_{l_i}, t_i + 2)$, then T_i can be scheduled on π at time $t_i + 1$ by copying every task in D_i on π.*

These properties are useful to design a list ordering algorithm for the duplication UET-UCT. We suppose that machines are identified by a unique integer and mean by first available machine for D-path D_i the machine with the smallest ID on which the D-path D_i and task T_i can be scheduled on at a given time t. Let's build a list ordering algorithm for the duplication UET-UCT from here. Let m_k be the k^{th} machine. Algorithm 9.1 gives a pseudo-code of a list ordering algorithm for the duplication UET-UCT: we did not describe how to check if T_i can be scheduled on machine m_k as it is pretty easy to code using an array of integer for each machine. This check consists in looking if the D-path can be executed on machine m_k using the properties described in Proposition 9.1 if the D-path is not empty, or to assure that machine m_k is not use at time t otherwise.

Algorithm 9.1 is pretty straightforward. We iterate over the tasks given by the list L and compute the D-path of the current task T_i. If the D-path is not empty, then we try to find a machine that can process T_i and its D-path. Line 12 uses the properties of Proposition 9.1 to check if T_i can be scheduled on machine m_k. If no machine can process T_i and its D-path D_i, we schedule task T_i two time-units after its latest predecessor on the first earliest available machine that can process it. While looking for such a machine, we might have to increase time t at which to schedule T_i if no machine is free at t.

Note that some optimal schedules cannot be built with Algorithm 9.1, an example of which is described in Example 9.2.

Example 9.2. *We consider the duplication UET-UCT instance described in Figure 9.1. Figure 9.3 presents two optimal schedules for this instance: each line corresponds to a machine, and each column to a time unit. The optimal solution of Figure 9.3a can be built by Algorithm 9.1, using the following list: $L = [0, 3, 4, 5, 6, 7, 1, 2]$ (note that list $L' = [0, 3, 5, 4, 6, 7, 1, 2]$ produces the same schedule).*

Algorithm 9.1: LIST algorithm

```
1  Function List(L: list of tasks)
2  begin
3      σ ← ∅
4      for i' ∈ [1, n] do
5          i ← L[i']
6          Compute the D-path D_i
7          found ← false
8          if D_i ≠ ∅ then
9              t ← max_{j∈Γ⁻(i)} s_j + 1
10             k ← 1
11             while ¬found ∧ k ≤ m do
12                 if D_i and T_i can both be scheduled on m_k then
13                     found ← true
14                     Schedule D_i on m_k in σ
15                     Schedule T_i at time t on m_k in σ
16                 end
17                 k ← k + 1
18             end
19         end
20         if ¬found then
21             t ← max_{j∈Γ⁻(i)} s_j + 2
22             while ¬found do
23                 k ← 1
24                 while ¬found ∧ k ≤ m do
25                     if m_k is free at time t then
26                         found ← true
27                         Schedule T_i at time t on m_k in σ
28                     end
29                     k ← k + 1
30                 end
31                 t ← t + 1
32             end
33         end
34     end
35     return σ
36 end
```

However, the optimal solution described in Figure 9.3b cannot be built using Algorithm 9.1. Indeed, the second machine cannot be saved by the algorithm as task T_4 appears necessarily before tasks T_6 and T_7 in the list as it precedes them. However, we can easily see that these two solutions are the same by machine symmetry.

Theorem 9.1. *For a given instance of the duplication UET-UCT, there exists a list L such that Algorithm 9.1 produces an optimal schedule.*

Proof. Consider an optimal schedule σ_{opt} such that all copies of a given task are necessarily in the D-path of another task. Such a schedule always exists: if T_i has several copies and at most one successor at time $s_i + 1$, then all other successors of

0	3	5	6
0	4	1	2
0	3	5	7

(A) An optimal schedule

0	3	5	6
0	3	5	7
0	4	1	2

(B) Another optimal schedule

FIGURE 9.3: Two optimal schedules of the duplication UET-UCT instance in Figure 9.1

T_i use the communication, meaning no copy other than the one on the machine with $j \in \Gamma^+(i)$ and $s_j = s_i + 1$ are useful and can therefore be removed from the schedule.

The list L is built pretty much following the timeline. While we build the list, we also maintain a schedule σ that would be built by L through Algorithm 9.1. We note $t_i(\sigma)$ the time at which task T_i would be scheduled by the list algorithm. The idea is to iterate over time and add all the tasks in the list (and update the schedule σ accordingly) by increasing number of machine's id. If an operation of scheduling task T_i by the list algorithm does not lead to the optimal schedule in such a case, it means that the solution cannot be reached, but it can be after a swap between two machines. When scanning tasks scheduled at time t, if $t_i(\sigma) < t$ then the task T_i must be postponed in the list, as adding it now would not lead to the same schedule. It arrives when there are copies of another task and a D-path is ongoing. Such a task T_i should not be added to L until $t_i(\sigma) = s_i$. □

9.2.2 Specialisation of the ORDER constraint for the duplication UET-UCT

In this subsection, we will see how to adapt the ORDER constraint to the duplication UET-UCT. We will especially see how we can consider other constraints' filtering in our propagator to improve on the filtering of the ORDER constraint.

The specification of the function f is such that it returns the time to which schedule task T_i following the list ordering algorithm (Algorithm 9.1). However, we can look a bit further. Indeed, we can also consider a schedule σ' that is built following the list L as well as the actual value of the assignment variables. As such, thanks to the filtering of other constraints, it might lead to postpone some tasks compared to what it would be without taking into account these values: in such a case, we have $t_i(\sigma) < t_i(\sigma')$. Whenever $t_i(\sigma) \neq t_i(\sigma')$ for a task T_i, then we can remove i from the current permutation variable as such an instantiation would contradict other constraints' filtering on assignment variables. Note that we can also enforce start variable s_i to be greater than $t_i(\sigma')$ as it will necessarily be the case: the solver will deduce it deeper in the tree following constraints' consistency. Algorithm 9.2 gives the propagator for the specialisation of the ORDER constraint to the duplication UET-UCT. We encoded the fact that a task must be after its predecessors in L inside this algorithm for performance purpose (instead of using an INVERSE constraint as well as a GENERALIZEDALLDIFFPREC constraint, as we did for the RCPSP).

The resulting CP model is the same as the one described in Figure 9.2, to which we only add the ORDER constraint as the filtering of the INVERSE and GENERALIZEDALLDIFFPREC constraints is done directly in the propagator (see Algorithm 9.2), contrary to what was done for the RCPSP.

Algorithm 9.2: Propagator for the ORDER constraint for the duplication UET-UCT

```
1  Function Propagate(X : variables, P : permutation variables)
2  begin
3      i ← 1
4      σ ← ∅
5      do
6          while |D(p_i)| = 1 do
7              Let m_k be the machine corresponding to t_{p_i}(σ)
8              Schedule D_i on m_k in σ
9              Schedule T_{p_i} at time t_{p_i}(σ) on m_k in σ
10             D(s_{p_i}) ← D(s_{p_i}) ∩ {t_{p_i}(σ)}
11             D(b_{p_i,k}) ← {1}
12             i ← i + 1
13         end
14         if i ≤ n then
15             Build σ' following σ, P and the assignment variables
16             for j ∈ D(p_i) do
17                 if Γ⁻(j) ⊈ {p_1, ..., p_{i-1}} ∨ t_j(σ) < s_j ∨ t_j(σ) ≠ t_j(σ') then
18                     D(p_i) ← D(p_i) \ {j}
19                 end
20             end
21         end
22     while |D(p_i)| = 1 ∧ i ≤ n
23 end
```

9.3 Experiments

9.3.1 Experimental protocol

For each instance, we compare the performance between the model described in Figure 9.2 and the model also using an ORDER constraint whose propagator is described in Algorithm 9.2. For the classic model described in Figure 9.2, we use the search heuristic *Smallest*, which selects the start variable with the smallest lower bound and branches on this value. This configuration will be referred to as SMALLEST. For the model also using an ORDER constraint whose propagator is described in Algorithm 9.2, the search we use branches on the current permutation variable p_{idx}. The selected value to branch on corresponds to the start variable with the smallest lower bound: $v \in D(p_{idx})$ such that $\underline{s_v} = \min(\{\underline{s_k} \mid k \in D(p_{idx})\})$. This configuration will be referred to as ORDER.

The instances we work on for the duplication UET-UCT are built as for the non-duplication UET-UCT in [Zin+10]: we took precedence graphs from the Standard Task Graph Set (STG) [Lab21], which is "a kind of benchmark for evaluation of multiprocessor scheduling algorithms". As in [Zin+10], we use the graphs from the STG as the precedence graphs in our UET-UCT instances, and fix the number of machines to 2, 3, 4 or 5. The instances have 50, 100 or 300 tasks, and the precedence graphs have various profile in terms of density, width, etc. In total, there are 2160 instances, 720 of size 50, 720 of size 100 and 720 of size 300.

The Choco-solver library [PFL17] was used in its 4.10.5 version. A time limit of 5 minutes was given for each configuration and each instance.

Every instance and piece of code that were used can be found in the GitHub repository [God21b].

9.3.2 Experimental results

First, note that we expect the experimental results of the Order configuration to be very similar to the ones of the Smallest configuration. Indeed, since tasks have unit processing times, the combinatorial gain of the Order configuration in comparison to the Smallest configuration should be smaller as for other scheduling problems as the horizon is n in the case of the UET-UCT. That said, the combinatorial from the assignment variables might still help the Order model to be better (as its combinatorial is not modified from these variables, in contrary to the model described in Figure 9.2).

The results are given in the form of tables, the definition of the lines of which were given in Section 8.3.2. Of course, here, the number of proofs is given for the proofs done within a time limit of 5 minutes (instead of 30 minutes for the RCPSP) as said in previous subsection. See that we removed "Nb times best" lines from the tables as we have only two configurations here: the number of times a configuration has 1.0 as objective ratio is the number of instances minus the sum of the times each configuration is strictly better. All results are given in Table 9.1, Table 9.2, Table 9.3 and Table 9.4.

	Smallest	Order
#Proofs	1239	**1272**
TimeToProof (ms)	**1295**	1919
Objective	1.0194	**1.0010**
Nb times strictly best	43	**547**

Table 9.1: Results of the benchmark on the 2160 UET-UCT instances

	Smallest	Order
#Proofs	345	**380**
TimeToProof (ms)	1055	**796**
Objective	1.021	**1.0019**
Nb times strictly best	19	**205**

Table 9.2: Results of the benchmark on the 720 UET-UCT instances of size 50

The results in Table 9.1 confirm our hypothesis that the gain of using the Order constraint should be smaller than for the RCPSP and the PMSPAUR. Indeed, the Order configuration does a bit more proofs than the Smallest configuration, while the latter is faster to do the proof when both configurations did the proof. We can also see that Smallest and Order configurations got the same objective value within the time limit for 1570 instances, which corresponds to two thirds of our benchmark. For the last third of the benchmark, we can see that, overall, the Order configuration

	Smallest	Order
#Proofs	**401**	399
TimeToProof (ms)	**1406**	2280
Objective	1.0244	**1.0009**
Nb times strictly best	20	**201**

Table 9.3: Results of the benchmark on the 720 UET-UCT instances of size 100

	Smallest	Order
#Proofs	**493**	**493**
TimeToProof (ms)	**1368**	2391
Objective	1.0125	**1.0001**
Nb times strictly best	4	**141**

Table 9.4: Results of the benchmark on the 720 UET-UCT instances of size 300

finds better solutions. Table 9.2, Table 9.3 and Table 9.4 allow us to see that the phenomenon is quite evenly distributed among all the benchmark and that the Order configuration is better than the Smallest configuration whatever the size of the instances of the benchmark. The mean time to proof of each configuration shows that, whenever both configurations did the proof, the Smallest configuration should be preferred as it is faster in average.

9.4 Conclusion

In this chapter, we first gave a way to improve on the filtering of assignment variables using additional variables and constraints. Then, we showed how to adapt the List Algorithm [Pin12] into a list ordering algorithm for the duplication UET-UCT. After that, we described how to embed this list ordering algorithm into a propagator, therefore specialising the Order constraint for the duplication UET-UCT. Finally, both models, the improved one and the improved one with the Order constraint and variables, are experimentally compared. In average, the Order model got better results, doing more proofs than its counterpart and finding strictly better solutions on almost a third of the benchmark. As expected, the results are even more mitigated than for the PMSPAUR and the RCPSP, but we can still see the interest of the Order constraint for scheduling problems: the combinatorial gain leads to a greater number of proofs or to better solutions within the time limit.

Conclusion

Contributions

The allDiffPrec constraint is defined as an allDifferent constraint with precedence between some variables. This constraint can be used in several contexts, such as examination timetabling or to enforce precedence in the permutation constraint. In Chapter 4, we explored the state-of-the-art filtering algorithms for the allDiffPrec constraint. More especially, we revisited the constraint by going deeper in the analysis of each state-of-the-art filtering algorithm, correcting the pseudo-code given by Bessiere et al. [Bes+11] into a working algorithm. A detailed review of the worst-case time complexities was also made. Then, we exploited a lemma from Bessiere et al. [Bes+11] in order to strengthen the filtering for the allDiffPrec constraint. We showed that this new filtering algorithm enforces bounds(Z) consistency or range consistency depending on the values on which it was applied (bounds only or all the domain). It is also able to detect inconsistent but bounds(Z) consistent values, considering holes within domains. We also extended the allDiffPrec constraint into the GeneralizedAllDiffPrec constraint, where the precedence are variables. We showed that bounds(Z) consistency for the GeneralizedAllDiffPrec constraint can be done in $O(n^3)$, and $O(n^2)$ (down a branch of the search tree). In Chapter 5, we discussed on implementation details of each filtering algorithm presented in Chapter 4. Particular attention was given to the implementation of data structures as well as the global organisation of the code into the propagator, with respect to potential needed prior filtering, to accelerate the speed of execution of the propagator. Finally, we presented an experimental protocol to compare the solving capacities of each implementation of the allDiffPrec constraint. We showed that, whenever domains are intervals, one should use the filtering algorithm from Bessiere et al. [Bes+11] (see Algorithm 4.5), whereas one should use our new filtering algorithm whenever domains are sets (and therefore can contain holes).

On a second part, we explored the idea to use list ordering algorithms within Constraint Programming. We first introduced in Chapter 6 the Order constraint, for which we gave a basic filtering scheme using permutation variables. The underlying idea is to introduce into constraint solvers reasoning behind list ordering algorithms and explore the search space composed of the set of all permutations. We discussed on the interest of this new constraint, which mostly relies on a reduction of the search space's size. One should therefore determine if such a reduction can indeed lead to greater solving efficiency, that is whenever the set of all permutations is smaller than the search space on decision variables of the basic model. For instance, for the RCPSP, the search space's size for the basic model is d^n with $d = Hor$. Therefore, using the Order constraint might get interesting whenever $d^n > n!$.

Then, we showed in Chapter 7 how to use a specialisation of the Order constraint to solve the Parallel Machine Scheduling Problem with Additional Unit Resources (PMSPAUR), which was the first problem for which we used the idea of list ordering

algorithm in CP. We also gave cutting rules specific to the PMSPAUR. We experimentally showed on random instances that the cutting rules as well as the application of the Order constraint for the PMSPAUR both increase the solving performance, leading to a greater number of proofs of optimality and better results in average for models based on the specialisation of the Order constraint.

In Chapter 8, we showed how to apply the Order constraint for the famous Resource-Constrained Project Scheduling Problem (RCPSP): we introduced the leftShifted constraint as a specialisation of the Order constraint for a set of cumulative and end-before-start precedence constraints. We showed experimentally that this approach led to great results thanks to the symmetry breaking rule it allows. The improved model for the RCPSP was also the occasion to use the GeneralizedAllDiffPrec constraint.

Finally, in Chapter 9, we presented a list ordering algorithm for the duplication Unit Execution Time-Unit Communication Time (UET-UCT) based on the notion of D-path [MH97]. This algorithm is of course very similar to the one of Munier and Hanen [MH97] and to the list algorithm presented by Pinedo [Pin12] for the non-duplication UET-UCT. This list ordering algorithm was then embedded in a propagator as a specialisation of the Order constraint for the duplication UET-UCT. We also showed how to improve the strength of inference on assignment variables for the classical model as well as our new one, for which we also improve on the global filtering by considering the filtering done by other constraints on the assignment variables. This improved model was experimentally compared with a similar version also using the Order constraint. We showed that the model based on the Order constraint does more proofs of optimality in average, but the experimental results were more mitigated for this problem as for the two previous problems, as will be discussed in next section.

Limits of the contributions

Experimental results on the model described in Figure 8.1 show that additional filtering for the allDiffPrec or the GeneralizedAllDiffPrec constraints is not always beneficial. Indeed, it seems that the few additional filtering offered by the BessiereEtAl and GodetBC implementations is not worth the reduction of speed of execution due to higher complexities. The decomposed version of the GeneralizedAllDiffPrec, being far more faster to run, yields the better overall results thanks to its speed and the strong impact of the current permutation variable's instantiation on the propagation phase. Therefore, one should pay attention when using the allDiffPrec or the GeneralizedAllDiffPrec constraints to select the appropriate implementation given the circumstances.

When it comes to the Order constraint and its specialisations to given problems, the limits are twofold. First, the induced code developments might sadly not be generalised or re-used to solve other problems. Despite the possibility for great practical performance, it also relies on the expertise on constraint solvers of users, which might be disincentive on a wider use of the Order constraint to solve problems with constraint programming. The specialisations of the constraint for the treated problem being hard-coded, the Order constraint can not really be used into languages such as MiniZinc [Net+07] or XCSP3 [Aud+20]. The need of constraint solvers expertise

to implement a specialisation of the Order constraint is the greater disincentive between these reasons as operations researchers and engineers are quite used to adapt existing methods to their problem.

On another hand, all problems are not efficiently solved using a specialisation of the Order constraint. Indeed, the case of the duplication UET-UCT is a great example: the experimental results are quite similar between the classical model as well as our new model, this new model based on the Order constraint still being better to find better solutions or to do proofs of optimality. Similarly, for the RCPSP, the application of the Order constraint was not as efficient as it was in the case of the PMSPAUR. Moreover, there might exist dominance rules or symmetry breaking rules dedicated to the problem: for instance, the left-shifted dominance rule we exploited for the RCPSP was already proposed by Chu and Stuckey [CS15], using an element constraint.

It shows that all problems do not have the same potential to be efficiently solved using the Order constraint, depending on the efficiency to filter the permutation variables as well as the reduction of the search space's size offered by the methodology.

Perspectives and further research

Perspectives of further research on the allDiffPrec and the GeneralizedAllDiffPrec constraints are multiple. First, experiments on the allDiffPrec constraint alone have shown to be conclusive on the efficiency of our new filtering rule whenever domains are not intervals. For the problem of examination timetabling, which would exactly be modelled with only one allDiffPrec constraint, we can conclude of the efficiency of the new filtering rule in such a case. However, as we saw in Chapter 8, where we used a GeneralizedAllDiffPrec constraint in the advanced model for the RCPSP, experimental results show that the decomposed version of the constraint is the most efficient one in this particular case. Of course, this is also due to the search heuristic. Therefore, further research should be led on the interaction of the different filtering algorithms for the allDiffPrec constraint with other propagators, with complementary study on the interaction with the search heuristic.

Remember that in most cases, the filtering algorithm of Bessiere et al. [Bes+11] (Algorithm 4.5) is as efficient as our new filtering rule and is faster: experimental results on the median time to proof show this. Therefore, another lead for further research would be a statistical study on the behaviour of both filtering rules (BessiereEtAl, GodetBC and GodetRC). Indeed, as in [Boi+13] for the allDifferent constraint, it might be relevant to have a good estimator of when our new filtering rule would be useful (compared to bounds(Z) consistency enforced by the algorithm of Bessiere et al. [Bes+11]). If such an estimator would exist, then we might reach a better overall trade-off between speed of execution and filtering strength.

The leads for further research on the Order constraint are legion. There are numerous iterative solution-building algorithms for problems, a great source of which can be found within approximation algorithms [Vaz03], and therefore there are as many use-cases for the Order constraint in CP. One should be careful that it is guaranteed that at least one optimal solution can be built by such an algorithm. We gave several applications on scheduling problems, but other kinds of problems might also be good case of application, such as packing problems, as they are similar to scheduling problems in many ways. For instance, the one-dimensional Bin Packing

problem [EC71] could be a case of application, using the First-Fit algorithm [EC71; GGU72]. Indeed, there exists an optimal packing that can be built with the First-Fit algorithm when the items are given in the right order within the list given in parameter to the First-Fit algorithm. It is not unthinkable that other kinds of problems might also be good cases of application of the Order constraint, as long as a list ordering algorithm that can build an optimal solution exists.

As said for the RCPSP (Chapter 8), using permutation variables with a specialisation of the Order constraint might not be very efficient alone, but can be with additional cutting rules or symmetry breaking rules. For instance, for the RCPSP, enforcing tasks to be sorted by increasing start time within the permutation gives great results. One can note that such a cutting rule is not applicable on the basic CP model (Figure 3.1). This is another way to use the Order constraint and improve on the global solving performance. The permutation variables are additional variables, *i.e.* not necessary to model the problem, and are as such additional possibilities for symmetry breaking and dedicated search heuristics, most especially because in the case of the Order constraint the permutation variables entirely decide the problem (they are decision variables).

Interestingly, this is not the first case of applications to scheduling problems. The idea to look for a list instead of directly looking for the optimal schedule was first explored by Zhou for the Job Shop Scheduling Problem (JSSP) [Zho97]. Note that the application of the Order constraint for the RCPSP (see Chapter 8) was also directly tested on the JSSP, as the RCPSP is a generalisation of the JSSP. The approach developed by Zhou shows better results for the JSSP. As such, further research should consist in implementing the approach of Zhou. This would lead to a fair comparison on the same solver, and better comprehension of Zhou's approach might give ideas to improve the model using the Order constraint for the RCPSP. Also note that Zinder et al. [Zin+10] developed a dedicated branch-and-bound algorithm in the case of the non-duplication UET-UCT that looks for an optimal list for the List Algorithm [Pin12]. Further research should also consists in comparing our results on the non-duplication UET-UCT to the ones of Zinder et al. [Zin+10] to see how our generic approach behaves in comparison to specific methods.

Finally, another lead for further research on the Order constraint can be found in explanations [Jus03]. Indeed, explanations for the cumulative constraint [Sch+09; Sch+11; SFS13] have proven to be very efficient to solve scheduling problems. As filtering on other permutation variables than the current one is pretty weak, if not non-existent, explanations might be of great help to filter other permutation variables than the current one, and differently than through the allDifferent constraint (or the allDiffPrec or GeneralizedAllDiffPrec constraints whenever such constraints might be useful).

The methodology developed for the Order constraint and its specialisations to solve different problems is a reminder that the expressiveness of constraint programming allows to implement dedicated solving techniques within constraint solvers. Introducing dedicated solving methods into constraint solvers might bring great performance in practice.

As a takeaway, we would like to bring the reader's attention on the hybridisation of constraint programming with other solving techniques. Indeed, the permissiveness and the expressiveness of constraint programming allows generally for the introduction of such techniques within propagators, allowing CP to benefit from the efficiency of such techniques. It led to great success stories in the past, as with explanations [Jus03] which is a technique from SAT solvers, or with Benders decomposition [Hoo07], or

with watch literals [GJM06; Gen13] which is another techniques from SAT solvers. It seems therefore to be a good idea to inspire from other solving techniques from other Combinatorial Optimisation fields and tools to improve on the practical efficiency of constraint solvers.

On code development

This thesis was the occasion to get a better comprehension of how a constraint solver works, in particular Choco solver. Whether for the developments for which we gave details in the thesis, or for other developments on Choco solver itself that were done, a particular attention was always given to the CPU time, testing different implementations of a same algorithm and see which one is the fastest in practice. Extensive work on data structures is never a waste of time, neither is any analysis on CPU time when one can afford it.

Bibliography

[AB93] Abderrahmane Aggoun and Nicolas Beldiceanu. "Extending CHIP in order to Solve Complex Scheduling and Placement Problems". In: *Mathl. Comput. Modelling* 17.7 (1993), pp. 57–273 (cit. on pp. 3, 40, 104, 145).

[AHU74] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974. ISBN: 0-201-00029-6 (cit. on p. 10).

[Aud+20] Gilles Audemard, Frédéric Boussemart, Christophe Lecoutre, Cédric Piette, and Olivier Roussel. "XCSP^3 and its ecosystem". In: *Constraints An Int. J.* 25.1-2 (2020), pp. 47–69 (cit. on pp. 124, 148).

[Bac94] Paul Bachmann. *Die Analytische Zahlentheorie*. 1894, pp. 3-1761-01022006–9 (cit. on p. 7).

[Bai15] Lijie Bai. "Train platforming problem in busy and complex railway stations". Theses. Ecole Centrale de Lille, Aug. 2015 (cit. on pp. 1, 143).

[BC02] Nicolas Beldiceanu and Mats Carlsson. "A New Multi-resource cumulatives Constraint with Negative Heights". In: *Principles and Practice of Constraint Programming - CP 2002, 8th International Conference, CP 2002, Ithaca, NY, USA, September 9-13, 2002, Proceedings*. Ed. by Pascal Van Hentenryck. Vol. 2470. Lecture Notes in Computer Science. Springer, 2002, pp. 63–79 (cit. on p. 105).

[BC93] Christian Bessière and Marie-Odile Cordier. "Arc-Consistency and Arc-Consistency Again". In: *Proceedings of the 11th National Conference on Artificial Intelligence. Washington, DC, USA, July 11-15, 1993*. Ed. by Richard Fikes and Wendy G. Lehnert. AAAI Press / The MIT Press, 1993, pp. 108–113 (cit. on p. 25).

[Bee06] Peter van Beek. "Backtracking Search Algorithms". In: *Handbook of Constraint Programming*. Ed. by Francesca Rossi, Peter van Beek, and Toby Walsh. Vol. 2. Foundations of Artificial Intelligence. Elsevier, 2006, pp. 85–134 (cit. on pp. 19, 23, 32, 34, 35).

[Bel+07] Nicolas Beldiceanu, Mats Carlsson, Sophie Demassey, and Thierry Petit. "Global Constraint Catalogue: Past, Present and Future". In: *Constraints An Int. J.* 12.1 (2007), pp. 21–62 (cit. on p. 28).

[Ber57] C Berge. "TWO THEOREMS IN GRAPH THEORY". In: *Proceedings of the National Academy of Sciences of the United States of America* 43.9 (Sept. 1957), pp. 842–844. ISSN: 0027-8424 (cit. on p. 15).

[Bes+02] Christian Bessière, Pedro Meseguer, Eugene C. Freuder, and Javier Larrosa. "On forward checking for non-binary constraint satisfaction". In: *Artif. Intell.* 141.1/2 (2002), pp. 205–224 (cit. on p. 27).

[Bes+05] Christian Bessière, Jean-Charles Régin, Roland H. C. Yap, and Yuanlin Zhang. "An optimal coarse-grained arc consistency algorithm". In: *Artif. Intell.* 165.2 (2005), pp. 165–185 (cit. on p. 25).

[Bes+11] Christian Bessiere, Nina Narodytska, Claude-Guy Quimper, and Toby Walsh. "The AllDifferent Constraint with Precedences". In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - 8th International Conference, CPAIOR 2011, Berlin, Germany, May 23-27, 2011. Proceedings*. Ed. by Tobias Achterberg and J. Christopher Beck. Vol. 6697. Lecture Notes in Computer Science. Springer, 2011, pp. 36–52 (cit. on pp. xiii, 3, 49–55, 57–60, 64, 67, 70, 71, 78, 123, 125, 146, 148, 149).

[Bes06] Christian Bessiere. "Constraint Propagation". In: *Handbook of Constraint Programming*. Ed. by Francesca Rossi, Peter van Beek, and Toby Walsh. Vol. 2. Foundations of Artificial Intelligence. Elsevier, 2006, pp. 29–83 (cit. on pp. 19, 20, 23, 25–27).

[BFR95] Christian Bessière, Eugene C. Freuder, and Jean-Charles Régin. "Using Inference to Reduce Arc Consistency Computation". In: *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, August 20-25 1995, 2 Volumes*. Morgan Kaufmann, 1995, pp. 592–599 (cit. on p. 25).

[BFR99] Christian Bessière, Eugene C. Freuder, and Jean-Charles Régin. "Using Constraint Metaknowledge to Reduce Arc Consistency Computation". In: *Artif. Intell.* 107.1 (1999), pp. 125–148 (cit. on p. 25).

[BH03] Christian Bessière and Pascal Van Hentenryck. "To Be or Not to Be ... a Global Constraint". In: *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*. Ed. by Francesca Rossi. Vol. 2833. Lecture Notes in Computer Science. Springer, 2003, pp. 789–794 (cit. on p. 28).

[Bia+19] Giovanni Lo Bianco, Xavier Lorca, Charlotte Truchet, and Gilles Pesant. "Revisiting Counting Solutions for the Global Cardinality Constraint". In: *J. Artif. Intell. Res.* 66 (2019), pp. 411–441 (cit. on p. 35).

[BK57] M. Beckman and T.C. Koopmans. "Assignment problems and the location of economic activities". In: *Econometrica* 25 (1957), pp. 53–76 (cit. on pp. 1, 143).

[Bla+19] J. Blazewicz, K.H. Ecker, E. Pesch, G. Schmidt, M. Sterna, and J. Weglarz. *Handbook on Scheduling: From Theory to Practice*. International Handbooks on Information Systems. Springer International Publishing, 2019. ISBN: 9783319998497 (cit. on p. 37).

[BLK83] Jacek Blazewicz, Jan Karel Lenstra, and A. H. G. Rinnooy Kan. "Scheduling subject to resource constraints: classification and complexity". In: *Discrete Applied Mathematics* 5.1 (1983), pp. 11–24 (cit. on pp. 40–42).

[Boi+13] Jérémie Du Boisberranger, Danièle Gardy, Xavier Lorca, and Charlotte Truchet. "When is it worthwhile to propagate a constraint? A probabilistic analysis of AllDifferent". In: *Proceedings of the 10th Meeting on Analytic Algorithmics and Combinatorics, ANALCO 2013, New Orleans, Louisiana, USA, January 6, 2013*. Ed. by Markus E. Nebel and Wojciech Szpankowski. SIAM, 2013, pp. 80–90 (cit. on pp. 27, 125, 148).

[Bou+04] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre, and Lakhdar Sais. "Boosting Systematic Search by Weighting Constraints". In: *Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS 2004, Valencia, Spain, August 22-27, 2004*. Ed. by Ramón López de Mántaras and Lorenza Saitta. IOS Press, 2004, pp. 146–150 (cit. on pp. 34, 99).

[BP00] Philippe Baptiste and Claude Le Pape. "Constraint Propagation and Decomposition Techniques for Highly Disjunctive and Highly Cumulative Project Scheduling Problems". In: *Constraints An Int. J.* 5.1/2 (2000), pp. 119–139 (cit. on p. 41).

[BR96] Christian Bessière and Jean-Charles Régin. "MAC and Combined Heuristics: Two Reasons to Forsake FC (and CBJ?) on Hard Problems". In: *Proceedings of the Second International Conference on Principles and Practice of Constraint Programming, Cambridge, Massachusetts, USA, August 19-22, 1996*. Ed. by Eugene C. Freuder. Vol. 1118. Lecture Notes in Computer Science. Springer, 1996, pp. 61–75 (cit. on p. 34).

[BR97] Christian Bessière and Jean-Charles Régin. "Arc Consistency for General Constraint Networks: Preliminary Results". In: *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes*. Morgan Kaufmann, 1997, pp. 398–404 (cit. on p. 25).

[Bru+98] Peter Brucker, Sigrid Knust, Arno Schoo, and Olaf Thiele. "A branch and bound algorithm for the resource-constrained project scheduling problem". In: *Eur. J. Oper. Res.* 107.2 (1998), pp. 272–288 (cit. on p. 41).

[Bru13] P. Brucker. *Scheduling Algorithms*. Springer Berlin Heidelberg, 2013. ISBN: 9783662030882 (cit. on p. 37).

[CC91] Jean-Yves Colin and Philippe Chrétienne. "C.P.M. Scheduling with Small Communication Delays and Task Duplication". In: *Oper. Res.* 39.4 (1991), pp. 680–684 (cit. on p. 44).

[CCT91] Jacqueline Chabrier, Jean-Jacques Chabrier, and Francois Trousset. "Résolution efficace d'un problème de satisfaction de Contraintes : Le Million de Reines". In: vol. 11th. May 1991 (cit. on p. 21).

[CGT15] Valentina Cacchiani, Laura Galli, and Paolo Toth. "A tutorial on non-periodic train timetabling and platforming problems". In: *EURO J. Transp. Logist.* 4.3 (2015), pp. 285–320 (cit. on pp. 1, 143).

[Cho+06] Chiu Wo Choi, Warwick Harvey, J. H. M. Lee, and Peter J. Stuckey. "Finite Domain Bounds Consistency Revisited". In: *AI 2006: Advances in Artificial Intelligence, 19th Australian Joint Conference on Artificial Intelligence, Hobart, Australia, December 4-8, 2006, Proceedings*. Ed. by Abdul Sattar and Byeong-Ho Kang. Vol. 4304. Lecture Notes in Computer Science. Springer, 2006, pp. 49–58 (cit. on p. 25).

[Chu+16] Geoffrey Chu, Peter J. Stuckey, Andreas Schutt, Thorsten Ehlers, Graeme Gange, and Kathryn Francis. *Chuffed, a lazy clause generation solver*. Department of Computing and Information Systems University of Melbourne, Australia. 2016 (cit. on pp. 3, 100, 145).

[Chu36a] Alonzo Church. "A note on the Entscheidungsproblem". In: *Journal of Symbolic Logic* 1.1 (1936), pp. 40–41 (cit. on p. 8).

[Chu36b] Alonzo Church. "An Unsolvable Problem of Elementary Number Theory". In: *American Journal of Mathematics* 58.2 (1936), pp. 345–363. ISSN: 00029327, 10806377 (cit. on p. 8).

[CJ98] Assef Chmeiss and Philippe Jégou. "Efficient Path-Consistency Propagation". In: *Int. J. Artif. Intell. Tools* 7.2 (1998), pp. 121–142 (cit. on p. 25).

[Coo71] Stephen A. Cook. "The Complexity of Theorem-Proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: Association for Computing Machinery, 1971, pp. 151–158. ISBN: 9781450374644 (cit. on p. 12).

[CR73] Stephen A. Cook and Robert A. Reckhow. "Time bounded random access machines". In: *Journal of Computer and System Sciences* 7.4 (1973), pp. 354–375. ISSN: 0022-0000 (cit. on p. 10).

[CS15] Geoffrey Chu and Peter J. Stuckey. "Dominance breaking constraints". In: *Constraints An Int. J.* 20.2 (2015), pp. 155–182 (cit. on pp. 111, 125, 149).

[EC71] Samuel Eilon and Nicos Christofides. "The Loading Problem". In: *Management Science* 17.5 (1971), pp. 259–268 (cit. on pp. 126, 149).

[EK72] Jack R. Edmonds and Richard M. Karp. "Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems". In: *J. ACM* 19.2 (1972), pp. 248–264 (cit. on p. 15).

[Eul36] Leonhard Euler. "Solutio problematis ad geometriam situs pertinentis". In: *Commentarii Academiae Scientiarum Imperialis Petropolitanae* 8 (1736), pp. 128–140 (cit. on p. 13).

[Fei+21] Anna Maria Feit, Mathieu Nancel, Maximilian John, Andreas Karrenbauer, Daryl Weir, and Antti Oulasvirta. "AZERTY amélioré: computational design on a national scale". In: *Commun. ACM* 64.2 (2021), pp. 48–58 (cit. on pp. 1, 143).

[Flo62] Robert W. Floyd. "Algorithm 97: Shortest path". In: *Commun. ACM* 5.6 (1962), p. 345 (cit. on p. 41).

[Fre97] Eugene C. Freuder. "In Pursuit of the Holy Grail". en. In: *Constraints* 2.1 (Apr. 1997), pp. 57–61. ISSN: 1572-9354 (cit. on pp. 1, 144).

[Fro+16] Aurélien Froger, Michel Gendreau, Jorge E. Mendoza, Éric Pinson, and Louis-Martin Rousseau. "Maintenance scheduling in the electricity industry: A literature review". In: *European Journal of Operational Research* 251.3 (2016), pp. 695–706. ISSN: 0377-2217 (cit. on pp. 1, 143).

[FW91] Eugene C. Freuder and Richard J. Wallace. "Selective relaxation for constraint satisfaction problems". In: *Third International Conference on Tools for Artificial Intelligence, TAI '91, San Jose, CA, USA, November 10-13, 1991*. IEEE Computer Society, 1991, pp. 332–339 (cit. on p. 27).

[Gay+15] Steven Gay, Renaud Hartert, Christophe Lecoutre, and Pierre Schaus. "Conflict Ordering Search for Scheduling Problems". In: *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*. Ed. by Gilles Pesant. Vol. 9255. Lecture Notes in Computer Science. Springer, 2015, pp. 140–148 (cit. on p. 34).

[GB65] Solomon W. Golomb and Leonard D. Baumert. "Backtrack Programming". In: *J. ACM* 12.4 (1965), pp. 516–524 (cit. on p. 27).

[Gen13] Ian P. Gent. "Optimal Implementation of Watched Literals and More General Techniques". In: *J. Artif. Intell. Res.* 48 (2013), pp. 231–251 (cit. on pp. 127, 150).

[GGU72] M. R. Garey, Ronald L. Graham, and Jeffrey D. Ullman. "Worst-Case Analysis of Memory Allocation Algorithms". In: *Proceedings of the 4th Annual ACM Symposium on Theory of Computing, May 1-3, 1972, Denver, Colorado, USA*. Ed. by Patrick C. Fischer, H. Paul Zeiger, Jeffrey D. Ullman, and Arnold L. Rosenberg. ACM, 1972, pp. 143–150 (cit. on pp. 126, 149).

[GHS15] Steven Gay, Renaud Hartert, and Pierre Schaus. "Simple and Scalable Time-Table Filtering for the Cumulative Constraint". In: *Principles and Practice of Constraint Programming - 21st International Conference, CP 2015, Cork, Ireland, August 31 - September 4, 2015, Proceedings*. Ed. by Gilles Pesant. Vol. 9255. Lecture Notes in Computer Science. Springer, 2015, pp. 149–157 (cit. on p. 105).

[GJ78] M. R. Garey and David S. Johnson. ""Strong" NP-Completeness Results: Motivation, Examples, and Implications". In: *J. ACM* 25.3 (1978), pp. 499–508 (cit. on pp. 42, 104).

[GJ79] M. R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. ISBN: 0-7167-1044-7 (cit. on p. 12).

[GJM06] Ian P. Gent, Christopher Jefferson, and Ian Miguel. "Watched Literals for Constraint Propagation in Minion". In: *Principles and Practice of Constraint Programming - CP 2006, 12th International Conference, CP 2006, Nantes, France, September 25-29, 2006, Proceedings*. Ed. by Frédéric Benhamou. Vol. 4204. Lecture Notes in Computer Science. Springer, 2006, pp. 182–197 (cit. on pp. 32, 127, 150).

[GM95] M. Gondran and M. Minoux. *Graphes et algorithmes*. Collection De la Direction des études et recherches d'Électricité de France. Eyrolles, 1995. ISBN: 978-2-212-01571-3 (cit. on p. 12).

[God+20] Arthur Godet, Xavier Lorca, Emmanuel Hebrard, and Gilles Simonin. "Using Approximation within Constraint Programming to Solve the Parallel Machine Scheduling Problem with Additional Unit Resources". In: *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020*. AAAI Press, 2020, pp. 1512–1519 (cit. on pp. 4, 42, 91).

[God19] Arthur Godet. *GitHub repository of Order for the PMSPAUR*. IMT Atlantique. 2019. URL: https://github.com/ArthurGodet/PMSPAUR-public (visited on 06/28/2021) (cit. on pp. 99, 100).

[God21a] Arthur Godet. *GitHub repository of Order for the RCPSP*. IMT Atlantique. 2021. URL: https://github.com/ArthurGodet/RCPSP-public (visited on 06/28/2021) (cit. on p. 108).

[God21b] Arthur Godet. *GitHub repository of Order for the UET-UCT*. IMT Atlantique. 2021. URL: https://github.com/ArthurGodet/UETUCT-order-public (visited on 06/28/2021) (cit. on p. 120).

[God21c] Arthur Godet. *GitHub repository on the AllDiffPrec constraint*. IMT Atlantique. 2021. URL: https://github.com/ArthurGodet/AllDiffPrec-public (visited on 06/28/2021) (cit. on p. 78).

[Göd31] Kurt Gödel. "Über formal unentscheidbare Sätze der Principia Mathematica und verwandter Systeme I". In: *Monatshefte für Mathematik und Physik* 38.1 (Dec. 1931), pp. 173–198. ISSN: 1436-5081 (cit. on p. 8).

[GP18] Samuel Gagnon and Gilles Pesant. "Accelerating Counting-Based Search". In: *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 15th International Conference, CPAIOR 2018, Delft, The Netherlands, June 26-29, 2018, Proceedings*. Ed. by Willem Jan van Hoeve. Vol. 10848. Lecture Notes in Computer Science. Springer, 2018, pp. 245–253 (cit. on p. 35).

[Gra+79] R.L. Graham, E.L. Lawler, J.K. Lenstra, and A.H.G.Rinnooy Kan. "Optimization and Approximation in Deterministic Sequencing and Scheduling: a Survey". In: *Discrete Optimization II*. Ed. by P.L. Hammer, E.L. Johnson, and B.H. Korte. Vol. 5. Annals of Discrete Mathematics. Elsevier, 1979, pp. 287–326 (cit. on pp. 38, 39).

[GRW08] B.L. Golden, S. Raghavan, and E.A. Wasil. *The Vehicle Routing Problem: Latest Advances and New Challenges*. Operations Research/Computer Science Interfaces Series. Springer US, 2008. ISBN: 9780387777788 (cit. on pp. 1, 143).

[GT85] Harold N. Gabow and Robert Endre Tarjan. "A Linear-Time Algorithm for a Special Case of Disjoint Set Union". In: *J. Comput. Syst. Sci.* 30.2 (1985), pp. 209–221 (cit. on pp. 57, 76).

[HA28] David Hilbert and Wilhelm Ackermann. *Grundzüge der theoretischen Logik*. ger. «Die »Grundlehren der mathematischen Wissenschaften. Berlin: Julius Springer, 1928 (cit. on p. 8).

[HC88] Pascal Van Hentenryck and Jean-Philippe Carillon. "Generality versus Specificity: An Experience with AI and OR Techniques". In: *Proceedings of the 7th National Conference on Artificial Intelligence, St. Paul, MN, USA, August 21-26, 1988*. Ed. by Howard E. Shrobe, Tom M. Mitchell, and Reid G. Smith. AAAI Press / The MIT Press, 1988, pp. 660–664 (cit. on p. 111).

[HE80] Robert M. Haralick and Gordon L. Elliott. "Increasing Tree Search Efficiency for Constraint Satisfaction Problems". In: *Artif. Intell.* 14.3 (1980), pp. 263–313 (cit. on pp. 27, 34).

[Heb+16] Emmanuel Hebrard, Marie-José Huguet, Nicolas Jozefowiez, Adrien Maillard, Cédric Pralet, and Gérard Verfaillie. "Approximation of the parallel machine scheduling problem with additional unit resources". In: *Discrete Applied Mathematics* 215 (2016), pp. 126–135 (cit. on pp. 42, 91, 92, 94, 95).

[Hen89] Pascal Van Hentenryck. *Constraint satisfaction in logic programming*. Logic programming. MIT Press, 1989. ISBN: 978-0-262-08181-8 (cit. on pp. 19, 27).

[HK73] John E. Hopcroft and Richard M. Karp. "An $n^{5/2}$ Algorithm for Maximum Matchings in Bipartite Graphs". In: *SIAM J. Comput.* 2.4 (1973), pp. 225–231 (cit. on pp. 9, 15, 64).

[HLM69] E. J. Hoffman, J. C. Loessi, and R. C. Moore. "Constructions for the Solution of the m Queens Problem". In: *Mathematics Magazine* 42.2 (1969), pp. 66–72. ISSN: 0025570X, 19300980 (cit. on p. 21).

[Hoa61] C. A. R. Hoare. "Algorithm 64: Quicksort". In: *Commun. ACM* 4.7 (July 1961), p. 321. ISSN: 0001-0782 (cit. on p. 10).

[Hoo07] John N. Hooker. "Planning and Scheduling by Logic-Based Benders Decomposition". In: *Oper. Res.* 55.3 (2007), pp. 588–602 (cit. on pp. 126, 150).

[HPV75] John E. Hopcroft, Wolfgang J. Paul, and Leslie G. Valiant. "On Time versus Space and Related Problems". In: *16th Annual Symposium on Foundations of Computer Science, Berkeley, California, USA, October 13-15, 1975*. IEEE Computer Society, 1975, pp. 57–64 (cit. on p. 10).

[HSD94] Pascal Van Hentenryck, Vijay A. Saraswat, and Yves Deville. "Design, Implementation, and Evaluation of the Constraint Language cc(FD)". In: *Constraint Programming: Basics and Trends, Châtillon Spring School, Châtillon-sur-Seine, France, May 16 - 20, 1994, Selected Papers*. Ed. by Andreas Podelski. Vol. 910. Lecture Notes in Computer Science. Springer, 1994, pp. 293–316 (cit. on p. 23).

[HT19] Djamal Habet and Cyril Terrioux. "Conflict history based search for constraint satisfaction problem". In: *Proceedings of the 34th ACM/SIGAPP Symposium on Applied Computing, SAC 2019, Limassol, Cyprus, April 8-12, 2019*. Ed. by Chih-Cheng Hung and George A. Papadopoulos. ACM, 2019, pp. 1117–1122 (cit. on p. 34).

[IK83] T. Ibaraki and N. Katoh. "On-line computation of transitive closures of graphs". In: *Information Processing Letters* 16.2 (1983), pp. 95–97. ISSN: 0020-0190 (cit. on p. 70).

[JK19] Maximilian John and Andreas Karrenbauer. "Dynamic Sparsification for Quadratic Assignment Problems". In: *Mathematical Optimization Theory and Operations Research - 18th International Conference, MOTOR 2019, Ekaterinburg, Russia, July 8-12, 2019, Proceedings*. Ed. by Michael Khachay, Yury Kochetov, and Panos M. Pardalos. Vol. 11548. Lecture Notes in Computer Science. Springer, 2019, pp. 232–246 (cit. on pp. 1, 143).

[Joh74] David S. Johnson. "Fast Algorithms for Bin Packing". In: *J. Comput. Syst. Sci.* 8.3 (1974), pp. 272–314 (cit. on p. 87).

[Jr63] J.E. Kelley Jr. "The Critical Path Method: Resources Planning and Scheduling". In: *Industrial Scheduling* 13 (1963) (cit. on p. 105).

[Jus03] Narendra Jussien. "The versatility of using explanations within constraint programming". Habilitation à diriger des recherches. Université de Nantes, Sept. 2003 (cit. on pp. 3, 126, 145, 150).

[Kan76] A.H.G. Rinnooy Kan. *Machine Scheduling Problems: Classification, complexity and computations*. Springer US, 1976. ISBN: 9789024718481 (cit. on p. 37).

[Kar72] Richard M. Karp. "Reducibility Among Combinatorial Problems". In: *Proceedings of a symposium on the Complexity of Computer Computations, held March 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York, USA*. Ed. by Raymond E. Miller and James W. Thatcher. The IBM Research Symposia Series. Plenum Press, New York, 1972, pp. 85–103 (cit. on p. 12).

[Kol96] Rainer Kolisch. "Serial and parallel resource-constrained project scheduling methods revisited: Theory and computation". In: *European Journal of Operational Research* 90.2 (1996), pp. 320–333. ISSN: 0377-2217 (cit. on pp. 105, 106).

[KP12] S. Kumar and R. Panneerselvam. "A Survey on the Vehicle Routing Problem and Its Variants". In: *Intelligent Information Management* 4.3 (2012), pp. 66–74 (cit. on pp. 1, 143).

[KS97] Rainer Kolisch and Arno Sprecher. "PSPLIB - A project scheduling problem library: OR Software - ORSEP Operations Research Software Exchange Program". In: *European Journal of Operational Research* 96.1 (1997), pp. 205–216. ISSN: 0377-2217 (cit. on p. 108).

[Kuh55] Harold W. Kuhn. "The Hungarian Method for the Assignment Problem". In: *Naval Research Logistics Quarterly* 2.1–2 (Mar. 1955), pp. 83–97 (cit. on p. 15).

[Lab21] Kasahara Laboratory. *Standard Task Graph Set (STG)*. 2021. URL: http://www.kasahara.cs.waseda.ac.jp/schedule/index.html (visited on 06/28/2021) (cit. on p. 119).

[Lan09] E. Landau. *Handbuch der Lehre von der Verteilung der Primzahlen*. 2 volumes. Reprinted by Chelsea, New York, 1953. Leipzig: Teubner, 1909 (cit. on p. 7).

[Lau78] Jean-Louis Laurière. "A Language and a Program for Stating and Solving Combinatorial Problems". In: *Artif. Intell.* 10.1 (1978), pp. 29–127 (cit. on p. 19).

[LBC12] Arnaud Letort, Nicolas Beldiceanu, and Mats Carlsson. "A Scalable Sweep Algorithm for the cumulative Constraint". In: *Principles and Practice of Constraint Programming - 18th International Conference, CP 2012, Québec City, QC, Canada, October 8-12, 2012. Proceedings*. Ed. by Michela Milano. Vol. 7514. Lecture Notes in Computer Science. Springer, 2012, pp. 439–454 (cit. on p. 105).

[LBH03] Christophe Lecoutre, Frédéric Boussemart, and Fred Hemery. "Exploiting Multidirectionality in Coarse-Grained Arc Consistency Algorithms". In: *Principles and Practice of Constraint Programming - CP 2003, 9th International Conference, CP 2003, Kinsale, Ireland, September 29 - October 3, 2003, Proceedings*. Ed. by Francesca Rossi. Vol. 2833. Lecture Notes in Computer Science. Springer, 2003, pp. 480–494 (cit. on p. 25).

[Lec96] M. Leconte. "A bounds-based reduction scheme for constraints of difference". In: *Proceedings of Second International Workshop on Constraint-based Reasoning (Constraint-96)*. 1996 (cit. on pp. 26, 52).

[LO00] François Laburthe and le projet OCRE. "Choco : implémentation du noyau d'un système de contraintes". In: *Actes des 6e Journées Nationales sur la résolution de Problèmes NP-Complets*. June 2000, pp. 151–165 (cit. on pp. 30, 31).

[Lóp+03] Alejandro López-Ortiz, Claude-Guy Quimper, John Tromp, and Peter van Beek. "A Fast and Simple Algorithm for Bounds Consistency of the AllDifferent Constraint". In: *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence, Acapulco, Mexico, August 9-15, 2003*. Ed. by Georg Gottlob and Toby Walsh. Morgan Kaufmann, 2003, pp. 245–250 (cit. on pp. 27, 54, 76).

[Mac77] Alan K. Mackworth. "Consistency in Networks of Relations". In: *Artif. Intell.* 8.1 (1977), pp. 99–118 (cit. on p. 24).

[MH12] Laurent Michel and Pascal Van Hentenryck. "Activity-Based Search for Black-Box Constraint Programming Solvers". In: *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems - 9th International Conference, CPAIOR 2012, Nantes, France, May 28 - June1, 2012. Proceedings*. Ed. by Nicolas Beldiceanu, Narendra Jussien, and Eric Pinson. Vol. 7298. Lecture Notes in Computer Science. Springer, 2012, pp. 228–243 (cit. on p. 34).

[MH86] Roger Mohr and Thomas C. Henderson. "Arc and Path Consistency Revisited". In: *Artif. Intell.* 28.2 (1986), pp. 225–233 (cit. on p. 25).

[MH97] Alix Munier and Claire Hanen. "Using Duplication for Scheduling Unitary Tasks on m Processors with Unit Communication Delays". In: *Theor. Comput. Sci.* 178.1-2 (1997), pp. 119–127 (cit. on pp. 44, 113, 115, 116, 124, 147).

[Mon74] Ugo Montanari. "Networks of constraints: Fundamental properties and applications to picture processing". In: *Inf. Sci.* 7 (1974), pp. 95–132 (cit. on p. 19).

[Mos+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. "Chaff: Engineering an Efficient SAT Solver". In: *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*. ACM, 2001, pp. 530–535 (cit. on p. 32).

[MS20] Andrea Mor and Maria Grazia Speranza. "Vehicle routing problems over time: a survey". In: *4OR* 18.2 (2020), pp. 129–149 (cit. on pp. 1, 143).

[MT00] Kurt Mehlhorn and Sven Thiel. "Faster Algorithms for Bound-Consistency of the Sortedness and the Alldifferent Constraint". In: *Principles and Practice of Constraint Programming - CP 2000, 6th International Conference, Singapore, September 18-21, 2000, Proceedings*. Ed. by Rina Dechter. Vol. 1894. Lecture Notes in Computer Science. Springer, 2000, pp. 306–319 (cit. on pp. 27, 70, 74).

[MV80] Silvio Micali and Vijay V. Vazirani. "An O(sqrt(|v|) |E|) Algorithm for Finding Maximum Matching in General Graphs". In: *21st Annual Symposium on Foundations of Computer Science, Syracuse, New York, USA, 13-15 October 1980*. IEEE Computer Society, 1980, pp. 17–27 (cit. on p. 18).

[Net+07] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. "MiniZinc: Towards a Standard CP Modelling Language". In: *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*. Ed. by Christian Bessiere. Vol. 4741. Lecture Notes in Computer Science. Springer, 2007, pp. 529–543 (cit. on pp. 124, 148).

[NT63] J. von Neumann and A.H. Taub. *John von Neumann Collected Works: Volume V - Design of Computers, Theory of Automata and Numerical Analysis*. Pergamon Press, 1963 (cit. on p. 10).

[Pap+94] Claude Le Pape, Philippe Couronné, Didier Vergamini, and Vincent Gosselin. "Time-versus-Capacity Compromises in Project Scheduling". In: *Proceedings of the Thirteenth Workshop of the U.K. Planning Special Interest Group*. 1994 (cit. on pp. 99, 107).

[Pap94] Christos H. Papadimitriou. *Computational complexity*. Addison-Wesley, 1994. ISBN: 978-0-201-53082-7 (cit. on p. 12).

[PF] Laurent Perron and Vincent Furnon. *OR-Tools*. Version 7.2. Google (cit. on pp. 3, 145).

[PFL17] Charles Prud'homme, Jean-Guillaume Fages, and Xavier Lorca. *Choco Documentation*. TASC - LS2N CNRS UMR 6241, COSLING S.A.S. 2017 (cit. on pp. 3, 35, 73, 78, 100, 108, 119, 145).

[Pin12] M.L. Pinedo. *Scheduling: Theory, Algorithms, and Systems*. SpringerLink : Bücher. Springer New York, 2012. ISBN: 9781461423614 (cit. on pp. 37, 113, 114, 121, 124, 126, 147, 149).

[PQZ12] Gilles Pesant, Claude-Guy Quimper, and Alessandro Zanarini. "Counting-Based Search: Branching Heuristics for Constraint Satisfaction Problems". In: *J. Artif. Intell. Res.* 43 (2012), pp. 173–210 (cit. on p. 35).

[Pra+14] Cédric Pralet, Gérard Verfaillie, Adrien Maillard, Emmanuel Hebrard, Nicolas Jozefowiez, Marie-José Huguet, Thierry Desmousceaux, Pierre Blanc-Paques, and Jean Jaubert. "Satellite Data Download Management with Uncertainty about the Generated Volumes". In: *Proceedings of the Twenty-Fourth International Conference on Automated Planning and Scheduling (ICAPS)*. Portsmouth, New Hampshire, USA, June 2014 (cit. on p. 41).

[Pru+14] Charles Prud'homme, Xavier Lorca, Rémi Douence, and Narendra Jussien. "Propagation engine prototyping with a domain specific language". In: *Constraints An Int. J.* 19.1 (2014), pp. 57–76 (cit. on pp. 3, 145).

[Pru14] Charles Prud'homme. "Contrôle de la propagation et de la recherche dans un solveur de contraintes. (Controlling propagation and search within a constraint solver)". PhD thesis. École des mines de Nantes, France, 2014 (cit. on pp. 3, 145).

[Pug98] Jean-Francois Puget. "A Fast Algorithm for the Bound Consistency of alldiff Constraints". In: *Proceedings of the Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98, July 26-30, 1998, Madison, Wisconsin, USA*. Ed. by Jack Mostow and Chuck Rich. AAAI Press / The MIT Press, 1998, pp. 359–366 (cit. on p. 52).

[Ray87] Victor J. Rayward-Smith. "UET scheduling with unit interprocessor communication delays". In: *Discret. Appl. Math.* 18.1 (1987), pp. 55–71 (cit. on p. 44).

[RBW06] Francesca Rossi, Peter van Beek, and Toby Walsh, eds. *Handbook of Constraint Programming*. Vol. 2. Foundations of Artificial Intelligence. Elsevier, 2006. ISBN: 978-0-444-52726-4 (cit. on p. 19).

[Ref04] Philippe Refalo. "Impact-Based Search Strategies for Constraint Programming". In: *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*. Ed. by Mark Wallace. Vol. 3258. Lecture Notes in Computer Science. Springer, 2004, pp. 557–571 (cit. on p. 34).

[Rég05] Jean-Charles Régin. "AC-*: A Configurable, Generic and Adaptive Arc Consistency Algorithm". In: *Principles and Practice of Constraint Programming - CP 2005, 11th International Conference, CP 2005, Sitges, Spain, October 1-5, 2005, Proceedings*. Ed. by Peter van Beek. Vol. 3709. Lecture Notes in Computer Science. Springer, 2005, pp. 505–519 (cit. on p. 25).

[Rég94] J.-C. Régin. "A Filtering Algorithm for Constraints of Difference in CSPs". In: *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI)*. Seattle, WA, USA, July 1994, pp. 362–367 (cit. on pp. 26, 31, 62, 96).

[Roy59] Bernard Roy. "Transitivité et connexité". In: *C. R. Acad. Sci. Paris* 249 (1959), pp. 216–218 (cit. on p. 41).

[SC06] Christian Schulte and Mats Carlsson. "Finite Domain Constraint Programming Systems". In: *Handbook of Constraint Programming*. Ed. by Francesca Rossi, Peter van Beek, and Toby Walsh. Vol. 2. Foundations of Artificial Intelligence. Elsevier, 2006, pp. 495–526 (cit. on pp. 19, 28–31).

[Sch+09] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark Wallace. "Why Cumulative Decomposition Is Not as Bad as It Sounds". In: *Principles and Practice of Constraint Programming - CP 2009, 15th International Conference, CP 2009, Lisbon, Portugal, September 20-24, 2009, Proceedings*. Ed. by Ian P. Gent. Vol. 5732. Lecture Notes in Computer Science. Springer, 2009, pp. 746–761 (cit. on pp. 3, 103, 126, 145, 150).

[Sch+11] Andreas Schutt, Thibaut Feydy, Peter J. Stuckey, and Mark G. Wallace. "Explaining the cumulative propagator". In: *Constraints An Int. J.* 16.3 (2011), pp. 250–282 (cit. on pp. 3, 103, 126, 145, 150).

[Sch70] Linus Schrage. "Solving Resource-Constrained Network Problems by Implicit Enumeration - Nonpreemptive Case". In: *Oper. Res.* 18.2 (1970), pp. 263–278 (cit. on p. 40).

[SFS13] Andreas Schutt, Thibaut Feydy, and Peter J. Stuckey. "Explaining Time-Table-Edge-Finding Propagation for the Cumulative Resource Constraint". In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 10th International Conference, CPAIOR 2013, Yorktown Heights, NY, USA, May 18-22, 2013. Proceedings*. Ed. by Carla P. Gomes and Meinolf Sellmann. Vol. 7874. Lecture Notes in Computer Science. Springer, 2013, pp. 234–250 (cit. on pp. 3, 103, 126, 145, 150).

[Sha04] Paul Shaw. "A Constraint for Bin Packing". In: *Principles and Practice of Constraint Programming - CP 2004, 10th International Conference, CP 2004, Toronto, Canada, September 27 - October 1, 2004, Proceedings*. Ed. by Mark Wallace. Vol. 3258. Lecture Notes in Computer Science. Springer, 2004, pp. 648–662 (cit. on p. 27).

[SKD95] Arno Sprecher, Rainer Kolisch, and Andreas Drexl. "Semi-active, active, and non-delay schedules for the resource-constrained project scheduling problem". In: *European Journal of Operational Research* 80.1 (1995), pp. 94–102. ISSN: 0377-2217 (cit. on p. 104).

[SS08] Christian Schulte and Peter J. Stuckey. "Efficient constraint propagation engines". In: *ACM Trans. Program. Lang. Syst.* 31.1 (2008), 2:1–2:43 (cit. on pp. 30, 31, 101).

[SZ15a] C. Schwindt and J. Zimmermann. *Handbook on Project Management and Scheduling Vol. 2*. International Handbooks on Information Systems. Springer International Publishing, 2015. ISBN: 9783319059150 (cit. on pp. 37, 39).

[SZ15b] C. Schwindt and J. Zimmermann. *Handbook on Project Management and Scheduling Vol.1*. International Handbooks on Information Systems. Springer International Publishing, 2015. ISBN: 9783319054438 (cit. on pp. 37, 39).

[Tar72] Robert Endre Tarjan. "Depth-First Search and Linear Graph Algorithms". In: *SIAM J. Comput.* 1.2 (1972), pp. 146–160 (cit. on p. 14).

[Tar75] Robert Endre Tarjan. "Efficiency of a Good But Not Linear Set Union Algorithm". In: *J. ACM* 22.2 (1975), pp. 215–225 (cit. on p. 76).

[Tom71] N. Tomizawa. "On some techniques useful for solution of transportation network problems". In: *Networks* 1.2 (1971), pp. 173–194 (cit. on p. 15).

[Tur36] Alan M. Turing. "On Computable Numbers, with an Application to the Entscheidungsproblem". In: *Proceedings of the London Mathematical Society* 2.42 (1936), pp. 230–265 (cit. on pp. 8, 9).

[Vaz03] Vijay V. Vazirani. *Approximation algorithms*. Springer, 2003. ISBN: 978-3-540-65367-7 (cit. on pp. 87, 125, 149).

[Vel93] B. Veltman. "Multiprocessor scheduling with communication delays". English. PhD thesis. Mathematics and Computer Science, 1993 (cit. on p. 44).

[VLS15] Petr Vilím, Philippe Laborie, and Paul Shaw. "Failure-Directed Search for Constraint-Based Scheduling". In: *Integration of AI and OR Techniques in Constraint Programming - 12th International Conference, CPAIOR 2015, Barcelona, Spain, May 18-22, 2015, Proceedings*. Ed. by Laurent Michel. Vol. 9075. Lecture Notes in Computer Science. Springer, 2015, pp. 437–453 (cit. on pp. 34, 108).

[War62] Stephen Warshall. "A Theorem on Boolean Matrices". In: *J. ACM* 9.1 (1962), pp. 11–12 (cit. on p. 41).

[Wat+19] Hugues Wattez, Christophe Lecoutre, Anastasia Paparrizou, and Sébastien Tabary. "Refining Constraint Weighting". In: *31st IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2019, Portland, OR, USA, November 4-6, 2019*. IEEE, 2019, pp. 71–77 (cit. on p. 34).

[Wol08] Armin Wolf. "Impact-Based Search in Constraint-based Scheduling". In: *38. Jahrestagung der Gesellschaft für Informatik, Beherrschbare Systeme - dank Informatik, INFORMATIK 2008, Munich, Germany, September 8-13, 2008, Band 2*. Ed. by Heinz-Gerd Hegering, Axel Lehmann, Hans Jürgen Ohlbach, and Christian Scheideler. Vol. P-134. LNI. GI, 2008, pp. 523–528 (cit. on p. 34).

[Zho97] Jianyang Zhou. "A Permutation-Based Approach for Solving the Job-Shop Problem". In: *Constraints An Int. J.* 2.2 (1997), pp. 185–213 (cit. on pp. 126, 149).

[Zin+10] Yakov Zinder, Bo Su, Gaurav Singh, and Ron Sorli. "Scheduling UET-UCT tasks: branch-and-bound search in the priority space". In: *Optimization and Engineering* 11 (2010), pp. 627–646 (cit. on pp. 114, 119, 126, 149).

Résumé long

Contexte

L'efficacité des processus industriels, des transports et de la société en général est importante, aujourd'hui plus que jamais. En effet, nos sociétés modernes se sont agrandies, de même que leur complexité sous-jacente. Par exemple, la logistique et le transport jouent un rôle clé dans le fonctionnement de nos sociétés. L'optimisation des processus sous-jacents, ainsi que de chaque activité indispensable aux services, est donc importante, tout particulièrement dans le contexte du changement climatique et de la transition sociétale. En effet, les économies d'énergie et d'argent sont importantes pour préserver au maximum la qualité de vie de chaque citoyen dans un tel contexte.

La Recherche Opérationnelle consiste précisément à modéliser un problème avec des variables et des contraintes mathématiques et à résoudre le problème modélisé. La recherche opérationnelle se concentre sur la résolution pratique de problèmes et utilise donc des techniques de résolution de plusieurs domaines. Différentes techniques existent pour résoudre ces problèmes, certaines plus faciles à implémenter en code que d'autres, certaines pouvant garantir l'inexistence de solutions ou l'optimalité d'une solution, certaines étant plus rapides en pratique que d'autres, etc. La diversité des techniques de résolution permet de sélectionner celle qui convient le mieux en fonction des contraintes du projet (compétences, temps de développement, temps de recherche d'une solution, capacité à prouver ou non l'optimalité, etc.).

Les problèmes dont l'objectif est de trouver une solution sont généralement appelés problèmes de satisfaction sous contraintes (CSP, pour *Constraint Satisfaction Problems* en anglais) et problèmes d'optimisation sous contraintes (COP, pour *Constraint Optimisation Problems* en anglais) chaque fois qu'il s'agit d'une fonction-objectif à minimiser ou à maximiser en plus du CSP. Ces problèmes ont en commun d'être combinatoires, c'est-à-dire que l'ensemble des possibilités est très large. Divers types de problèmes sont modélisés en tant que CSP et COP. Pour n'en citer que quelques-uns, cela va de la planification de la maintenance dans l'industrie électrique [Fro+16], aux problèmes logistiques de transport et de livraison (généralement appelés *Vehicule Routing Problems* (VRP) en anglais [GRW08; KP12; MS20]), ou au problème d'horairisation de circulations ferroviaires, qui consiste à attribuer une heure d'arrivée et une heure de départ, ainsi qu'un parcours sur les rails, à chaque train pour une gare donnée [Bai15; CGT15], ou encore la conception du nouveau clavier français [JK19; Fei+21] (modélisée comme une spécialisation du problème d'affectation quadratique [BK57]).

Dans le cadre de cette thèse, nous nous concentrons sur une technique de résolution appelée Programmation par Contraintes (PPC), qui est un paradigme déclaratif, c'est-à-dire que l'utilisateur spécifie le problème dans un langage de contraintes, interprétable par un logiciel appelé solveur de contraintes (ou simplement solveur), dont la tâche est de résoudre le problème. Ce comportement "boîte noire" existe depuis le début du domaine et dans l'esprit des chercheurs : « la programmation par contraintes représente l'une des approches les plus proches que l'informatique ait faites du Saint

Graal de la programmation : l'utilisateur pose le problème, l'ordinateur le résout » comme l'a déclaré Freuder dans [Fre97]. En effet, la PPC se distingue d'autres techniques telles que la Programmation Linéaire par la grande expressivité qu'elle permet pour les contraintes, ainsi que par la possibilité d'intégrer dans les solveurs de contraintes des algorithmes et des techniques d'autres domaines. Malheureusement, cette grande expressivité et permissivité de la PPC conduit à des solveurs de contraintes complexes, qui nécessitent une certaine expertise afin de modéliser et d'exprimer efficacement un problème et de le résoudre tout aussi efficacement. Ce comportement peut être vu comme une "boîte blanche" : le solveur permet beaucoup de choses, mais il nécessite une expertise pour le configurer efficacement. Dans cette thèse, nous serons plus proches du comportement "boîte blanche" que de celui "boîte noire".

Pour résoudre des problèmes en PPC, il faut d'abord modéliser le problème que l'on souhaite résoudre au moyen de variables et de contraintes. Par exemple, supposons que vous souhaitiez planifier la maintenance d'une flotte de trains. Vous disposez d'un ensemble de tâches, chacune correspondant à une activité de maintenance. L'objectif de votre problème est de décider quand traiter chacune des tâches, compte tenu des contraintes de ressources (deux trains ne peuvent pas occuper le même rail en même temps, horaire des ouvriers qualifiés, etc.) et des contraintes de planification (les trains arrivent et quittent le centre de maintenance à une heure précise pour respecter leur planning commercial par exemple). En considérant que les tâches ont une durée fixe pour être traitées, les variables de début des tâches définissent une solution du problème si aucune des contraintes n'est violée. Le modélisateur peut également préciser comment l'espace de recherche sera exploré : nous considérons que cela fait partie de la phase de modélisation. Une fois la phase de modélisation terminée, le solveur commence à explorer l'espace de recherche, à la recherche de solutions. L'ensemble des affectations de valeurs aux variables définies dans cette phase constitue l'espace de recherche.

Le solveur alterne entre deux phases : propagation et branchement. Le rôle de la propagation est d'inférer que certaines affectations de valeurs aux variables ne peuvent pas conduire à une solution du problème compte tenu des définitions des contraintes. Par exemple, s'il a déjà été décidé que la maintenance électrique d'un train donné se ferait de t_1 à t_2 et qu'il existe des machines pour faire la maintenance électrique uniquement pour un train à la fois, alors le moteur de propagation peut appliquer que les autres tâches de maintenance électrique des autres trains se terminent avant t_1 ou commencent après t_2. Cette phase de propagation peut conduire à de grandes coupures dans l'espace de recherche et ne doit donc pas être sous-estimée. Cependant, ces inférences ont un certain coût algorithmique, généralement plus l'inférence est forte, plus il est coûteux de la trouver. Dans cette thèse, nous nous intéresserons à ce compromis intéressant.

Chaque fois que les règles d'inférence des contraintes ne peuvent plus trouver d'affectations défaillantes, soit toutes les variables sont instanciées et une solution a été trouvée, soit il y a une contradiction entre les inférences des contraintes et nous devons revenir à un précédent état valide de la recherche, soit il y a indécision (il existe des variables non instanciées sans qu'aucune contrainte ne soit violée). Dans ce troisième cas, un branchement est effectué, c'est-à-dire le solveur prend une décision (par exemple la maintenance électrique du train A sera effectuée à l'instant t_1) et propage cette décision. La contradiction de cette décision sera également explorée. C'est pourquoi la Programmation par Contraintes est catégorisée comme méthode de recherche complète : elle explore toutes les possibilités et peut donc garantir l'inexistence de solutions

ou l'optimalité d'une solution (par exemple, elle peut garantir qu'il n'existe pas de planning plus court que celui trouvé). Effectuer les bons branchements en premier peut s'avérer crucial pour la performance.

En effet, dans son intervention lors de l'ACP Research Excellence Award (CP'13)[1], Jean-Charles Régin explique que prêter attention à la phase de modélisation, à l'exploration de l'espace de recherche et à l'implémentation (notamment des structures de données) est important, le gain potentiel d'efficacité suivant cet ordre et étant inverse à la probabilité de succès. C'est-à-dire que parmi ces trois phases, le gain d'efficacité le plus élevé peut être trouvé dans la phase de modélisation, mais il y a in fine peu de chances d'obtenir une amélioration de l'efficacité lors de la résolution.

Un autre point d'attention, bien que pas vraiment entre les mains des modélisateurs et des utilisateurs de solveurs, est le moteur de propagation : la manière dont il est codé et articulé peut avoir une grande importance en pratique lors de la résolution de problèmes, comme l'ont montré Prud'Homme et al. [Pru14; Pru+14]. Plus particulièrement, une grande connaissance d'un solveur et de son fonctionnement interne peut conduire à de meilleures performances en pratique car la mise en œuvre sera adaptée aux spécificités du solveur. Pour toutes ces raisons, nous accordons une attention particulière à l'implémentation tout au long de cette thèse.

Dans cette thèse, nous explorons différents aspects présentés précédemment. Nous nous concentrons principalement sur l'amélioration des performances de résolution en pratique des problèmes d'ordonnancement. Plus particulièrement, nous exploitons la grande expressivité de la PPC pour introduire dans les solveurs de contraintes d'autres techniques issues de l'optimisation combinatoire. Cela s'est avéré très efficace dans le passé : l'utilisation d'explications (une technique issue des solveurs SAT) [Jus03; Sch+09; Sch+11; SFS13] a conduit à des résultats impressionnants sur les problèmes d'ordonnancement classiques, et est un atout important de la PPC pour résoudre efficacement les problèmes d'ordonnancement. Malheureusement, il faut beaucoup d'efforts de développement pour interfacer les explications avec le moteur de propagation du solveur (comme cela a été fait dans Choco [PFL17]) ou pour les prendre en compte nativement (comme pour les solveurs LCG (pour *Lazy-Clause Generation*) tels que Chuffed [Chu+16] ou OR-Tools [PF], également appelés solveurs CP-SAT). De plus, les explications peuvent rencontrer des difficultés : par exemple, la version décomposée expliquée de la contrainte CUMULATIVE [AB93] peut consommer une très grande quantité de mémoire si le planning peut être construit sur une longue période de temps. Cela exclut l'utilisation d'explications dans des environnements à mémoire limitée, où les explications pourraient perdre en efficacité car certaines clauses sont oubliées (ce qui est fait par les solveurs LCG dans des environnements à mémoire limitée). Pour ces raisons, nous nous concentrons sur d'autres techniques, qui sont complémentaires aux explications.

Les contributions de cette thèse sont doubles. Dans une première partie, nous nous concentrons sur la contrainte ALLDIFFPREC, qui est la composition d'une contrainte ALLDIFFERENT et de contraintes de précédence. Nous faisons une revue complète de l'état-de-l'art sur cette contrainte, corrigeant un algorithme de filtrage, et proposons une nouvelle règle de filtrage. Dans une seconde partie, nous nous intéressons à l'utilisation d'algorithmes de liste ordonnée en PPC pour résoudre des problèmes

[1] « Des solutions simples à des problèmes complexes », Jean-Charles Régin. ACP Research Excellence Award, CP2013(Uppsala, Suède).

d'ordonnancement. Plus particulièrement, nous y développons comment ajouter des variables de permutation et les filtrer peut amener à de meilleures performances de résolution. Pour ceci, nous spécifions ce processus sur plus problèmes, montrant la généricité de l'approche ainsi qu'un panel de performances, dont la variabilité montre les limites de l'approche.

La contrainte allDiffPrec

La contrainte allDiffPrec est définie comme une contrainte allDifferent avec des précédences entre certaines variables. Cette contrainte peut être utilisée dans plusieurs contextes, tels que la planification d'examen ou pour imposer des précédences dans la contrainte permutation. Dans le Chapitre 4, nous explorons et décrivons l'ensemble des algorithmes de filtrage état-de-l'art pour la contrainte allDiffPrec. Plus particulièrement, nous revisitons la contrainte en approfondissant l'analyse de chaque algorithme de filtrage, en corrigeant le pseudo-code donné par Bessière et al. [Bes+11] en un algorithme fonctionnel. Un examen détaillé des complexités temporelles dans le pire cas est également effectué. Ensuite, nous utilisons un lemme de Bessière et al. [Bes+11] afin de renforcer le filtrage de la contrainte allDiffPrec. Nous démontrons que ce nouvel algorithme de filtrage applique la cohérence de bornes (*bounds(Z) consistency* en anglais) ou la *range consistency* en fonction des valeurs sur lesquelles il est appliqué (bornes uniquement ou tout le domaine). Ce nouvel algorithme est également capable de détecter des valeurs incohérentes mais cohérentes aux bornes, en tenant compte des trous dans les domaines. Nous étendons également la contrainte allDiffPrec à la contrainte GeneralizedAllDiffPrec, où les précédences sont des variables. Nous démontrons que la cohérence aux bornes pour la contrainte GeneralizedAllDiffPrec peut être effectuée en $O(n^3)$, et en $O(n^2)$ le long d'une branche de l'arbre de recherche.

Dans le Chapitre 5, nous discutons des détails d'implémentation de chaque algorithme de filtrage présenté dans le Chapitre 4. Une attention toute particulière est apportée à l'implémentation des structures de données ainsi qu'à l'organisation globale du code dans le propagateur, notamment vis-à-vis des potentiels filtrages préalables nécessaires, pour accélérer la vitesse d'exécution du propagateur. Enfin, nous présentons un protocole expérimental pour comparer les capacités de résolution de chaque implémentation de la contrainte allDiffPrec. Nous montrons que, chaque fois que les domaines sont des intervalles, il faut utiliser l'algorithme de filtrage de Bessière et al. [Bes+11] (voir Algorithme 4.5), alors qu'il faudrait utiliser notre nouvel algorithme de filtrage chaque fois que les domaines sont des ensembles (et peuvent donc contenir des trous).

Introduction d'algorithmes de liste ordonnée en PPC

Dans cette seconde partie, nous explorons l'idée d'utiliser des algorithmes de liste ordonnée en programmation par contraintes. Nous introduisons d'abord dans le Chapitre 6 la contrainte Order, pour laquelle nous donnons un schéma de filtrage de base utilisant des variables de permutation. L'idée sous-jacente est d'introduire dans les solveurs de contraintes le raisonnement derrière les algorithmes de liste ordonnée et d'explorer l'espace de recherche composé de l'ensemble de toutes les permutations. Nous discutons de l'intérêt de cette nouvelle contrainte, qui repose principalement sur une réduction de la taille de l'espace de recherche. Il faut donc déterminer si une telle réduction peut effectivement conduire à une plus grande efficacité de résolution,

c'est-à-dire à chaque fois que l'ensemble de toutes les permutations est plus petit que l'espace de recherche sur les variables de décision du modèle de base. Par exemple, pour le RCPSP, la taille de l'espace de recherche pour le modèle de base est Hor^n. Par conséquent, l'utilisation de la contrainte Order peut devenir intéressante chaque fois que $Hor^n > n!$. Notons que le filtrage assez "faible" sur les variables de permutation peut entraîner une résolution moins efficace.

Ensuite, nous montrons dans le Chapitre 7 comment utiliser une spécification de la contrainte Order pour résoudre le problème d'ordonnancement de machines parallèles avec des ressources unitaires supplémentaires (PMSPAUR pour *Parallel Machine Scheduling Problem with Additional Unit Resources* en anglais), qui était le premier problème pour lequel nous avons utilisé l'idée d'algorithme de liste ordonnée en PPC. Nous donnons également des règles de coupe spécifiques au PMSPAUR. Nous montrons expérimentalement sur des instances aléatoires que les règles de coupe ainsi que l'application de la contrainte Order pour le PMSPAUR augmentent toutes les deux les performances de résolution, conduisant à un plus grand nombre de preuves d'optimalité et à de meilleurs résultats en moyenne pour les modèles basés sur la spécification de la contrainte Order.

Dans le Chapitre 8, nous montrons comment appliquer la contrainte Order pour le célèbre problème de gestion de projet à contraintes de ressources (RCPSP, pour *Resource-Constrained Project Scheduling Problem* en anglais) : nous introduisons la contrainte leftShifted comme une spécification de la contrainte Order pour un ensemble de contraintes cumulative et de contraintes de précédence de type fin-avant-début. Nous montrons expérimentalement que cette approche conduit à d'excellents résultats grâce à la règle de cassage de symétrie qu'elle permet. Le modèle amélioré pour le RCPSP a également été l'occasion d'utiliser la contrainte GeneralizedAllDiffPrec.

Enfin, dans le Chapitre 9, nous présentons un algorithme de liste ordonnée pour le problème de planification de tâches unitaires sur machines parallèles avec temps de communication unitaires (UET-UCT, pour *Unit Execution Time-Unit Communication Time* en anglais) avec duplication, basé sur la notion de D-chemin [MH97]. Cet algorithme est bien entendu très similaire à celui de Munier et Hanen [MH97] et à l'algorithme de liste présenté par Pinedo [Pin12] pour l'UET-UCT sans duplication. Cet algorithme de liste ordonnée est ensuite intégré dans un propagateur en tant que spécification de la contrainte Order pour l'UET-UCT avec duplication. Nous montrons également comment améliorer la force de filtrage sur les variables d'affectation pour le modèle classique ainsi que notre nouveau modèle, pour lequel nous améliorons également le filtrage global en prenant en compte le filtrage effectué par d'autres contraintes sur les variables d'affectation. Ce modèle amélioré est comparé expérimentalement à une version similaire utilisant également la contrainte Order. Nous montrons que le modèle basé sur la contrainte Order fait plus de preuves d'optimalité en moyenne, mais les résultats expérimentaux sont plus mitigés pour ce problème que pour les deux problèmes précédents, comme nous le verrons dans la section suivante.

Conclusion

Les résultats expérimentaux sur le modèle décrit dans la Figure 8.1 montrent qu'un filtrage supplémentaire pour les contraintes allDiffPrec ou GeneralizedAllDiffPrec n'est pas toujours bénéfique. En effet, il semble que les quelques filtrages supplémentaires offerts par les implémentations BessiereEtAl et GodetBC ne valent pas la réduction de vitesse d'exécution en raison de complexités

plus élevées. La version décomposée de la contrainte GeneralizedAllDiffPrec, étant beaucoup plus rapide à exécuter, donne les meilleurs résultats globaux grâce à sa vitesse et au fort impact de l'instanciation de la variable de permutation actuelle sur la phase de propagation. Par conséquent, il convient de faire attention lors de l'utilisation des contraintes allDiffPrec ou GeneralizedAllDiffPrec à sélectionner l'implémentation appropriée compte tenu des circonstances.

En ce qui concerne la contrainte Order et ses spécifications pour des problèmes donnés, les limites sont doubles. Premièrement, les développements de code induits pourraient malheureusement ne pas être généralisés ou réutilisés pour résoudre d'autres problèmes. Malgré la possibilité de grandes performances en pratique, l'utilisation de la contrainte Order repose également sur l'expertise des utilisateurs sur les solveurs de contraintes, ce qui pourrait décourager une utilisation plus large de la contrainte Order pour résoudre des problèmes avec la programmation par contraintes. Les spécifications de la contrainte pour le problème traité étant codées "en dur", la contrainte Order ne peut pas vraiment être utilisée dans des langages tels que MiniZinc [Net+07] ou XCSP3 [Aud+20]. Le besoin d'expertise en solveurs de contraintes pour implémenter une spécification de la contrainte Order reste le plus dissuasif car les chercheurs et ingénieurs en recherche opérationnelle sont habitués à adapter les méthodes existantes à leur problème.

D'un autre côté, tous les problèmes ne sont pas résolus efficacement en utilisant une spécification de la contrainte Order. En effet, le cas de l'UET-UCT avec duplication est un bel exemple : les résultats expérimentaux sont assez similaires entre le modèle classique et notre nouveau modèle, ce nouveau modèle basé sur la contrainte Order étant tout de même meilleur pour trouver de meilleures solutions ou faire des preuves d'optimalité. De même, pour le RCPSP, l'application de la contrainte Order n'a pas été aussi efficace que dans le cas du PMSPAUR. Cela montre que tous les problèmes n'ont pas le même potentiel pour être résolus efficacement en utilisant la contrainte Order, en fonction de l'efficacité à filtrer les variables de permutation ainsi que de la réduction de la taille de l'espace de recherche offerte par l'approche.

Les perspectives de recherches ultérieures sur les contraintes allDiffPrec et GeneralizedAllDiffPrec sont multiples. Premièrement, les expériences sur la contrainte allDiffPrec seule se sont avérées concluantes sur l'efficacité de notre nouvelle règle de filtrage lorsque les domaines ne sont pas des intervalles. Pour le problème de planification d'examens, qui serait exactement modélisé avec une simple contrainte allDiffPrec, on peut conclure à l'efficacité de la nouvelle règle de filtrage dans un tel cas. Cependant, comme nous l'avons vu au Chapitre 8, où nous avons utilisé une contrainte GeneralizedAllDiffPrec dans le modèle avancé pour le RCPSP, les résultats expérimentaux montrent que la version décomposée de la contrainte est la plus efficace dans ce cas particulier. Bien sûr, cela est également dû à l'heuristique de recherche. Par conséquent, des recherches supplémentaires devraient être menées sur l'interaction des différents algorithmes de filtrage pour la contrainte allDiffPrec avec d'autres propagateurs, avec une étude complémentaire sur l'interaction avec l'heuristique de recherche.

Rappelons que dans la plupart des cas, l'algorithme de filtrage de Bessière et al. [Bes+11] (Algorithm 4.5) est aussi efficace que notre nouvelle règle de filtrage et est plus rapide : les résultats expérimentaux sur le temps médian pour faire la preuve le montrent. Par conséquent, une autre piste de recherche serait une étude statistique sur le comportement des deux règles de filtrage (BessiereEtAl, GodetBC et GodetRC). En effet, comme dans [Boi+13] pour la contrainte allDifferent, il pourrait être pertinent d'avoir un bon estimateur du moment où notre nouvelle règle

de filtrage serait utile par rapport à la cohérence des bornes effectuée par l'algorithme de Bessière et al. [Bes+11]. Si un tel estimateur existait, alors nous pourrions atteindre un meilleur compromis global entre la vitesse d'exécution et la force de filtrage.

Les pistes pour des recherches plus poussées sur la contrainte ORDER sont légion. Il existe de nombreux algorithmes itératifs de construction de solutions pour les CSP et COP, dont une grande source peut être trouvée dans les algorithmes d'approximation [Vaz03], et donc il y a autant de cas d'utilisation pour la contrainte ORDER en PPC. Il faut veiller à ce qu'il soit garanti qu'au moins une solution optimale puisse être construite par un tel algorithme. Nous avons donné plusieurs applications pour des problèmes d'ordonnancement, mais d'autres types de problèmes pourraient également être de bons cas d'application, tels que les problèmes de *packing*, car ils sont similaires aux problèmes d'ordonnancement à bien des égards. Par exemple, le problème de Bin Packing unidimensionnel [EC71] pourrait être un cas d'application, en utilisant l'algorithme First-Fit [EC71; GGU72]. En effet, il existe un *packing* optimal qui peut être construit avec l'algorithme First-Fit lorsque les éléments sont donnés dans le bon ordre au sein de la liste donnée en paramètre à l'algorithme First-Fit. Il n'est pas impensable que d'autres types de problèmes puissent également être de bons cas d'application de la contrainte ORDER, tant qu'il existe un algorithme de liste ordonnée qui peut construire une solution optimale.

Comme indiqué pour le RCPSP (Chapitre 8), l'utilisation de variables de permutation avec une spécification de la contrainte ORDER peut ne pas être très efficace seule, mais peut l'être avec des règles supplémentaires de coupe ou de cassage de symétrie. Par exemple, pour le RCPSP, imposer que les tâches soient triées par heure croissante de début dans la permutation donne d'excellents résultats. On peut noter qu'une telle règle de cassage de symétrie n'est pas applicable sur le modèle de base CP (Figure 3.1). C'est une autre façon d'utiliser la contrainte ORDER et d'améliorer les performances de résolution globales. Les variables de permutation sont des variables supplémentaires, c'est-à-dire non indispensables pour modéliser le problème, et sont en tant que telles des possibilités supplémentaires de cassage de symétrie et d'heuristiques de recherche dédiées, plus particulièrement parce que, dans le cas de la contrainte ORDER, les variables de permutation décident entièrement le problème (ce sont des variables de décision).

Fait intéressant, ce n'est pas le premier cas d'applications à des problèmes d'ordonnancement. L'idée de rechercher une liste au lieu de rechercher directement le planning optimal a été explorée pour la première fois par Zhou pour le *Job Shop Scheduling Problem* (JSSP) [Zho97]. Notons que l'application de la contrainte ORDER pour le RCPSP (voir Chapitre 8) a également été directement testée sur le JSSP, car le RCPSP est une généralisation du JSSP. L'approche développée par Zhou montre de meilleurs résultats pour le JSSP. A ce titre, d'autres recherches devraient consister à implémenter l'approche de Zhou. Cela conduirait à une comparaison équitable sur le même solveur, et une meilleure compréhension de l'approche de Zhou pourrait donner des idées pour améliorer le modèle utilisant la contrainte ORDER pour le RCPSP. De manière similaire, le fait de ne rechercher que des plannings *left-shifted* a également déjà été mentionné pour le RCPSP par Chu et Stuckey [CS15], leur implémentation étant basée sur une contrainte ELEMENT. Notons également que Zinder et al. [Zin+10] ont développé un algorithme par séparation et évaluation (*branch and bound algorithm* en anglais) dédié dans le cas de l'UET-UCT sans duplication qui recherche une liste optimale pour l'algorithme de liste [Pin12]. Des recherches supplémentaires devraient également consister à comparer nos résultats sur l'UET-UCT sans duplication à ceux de Zinder et al. [Zin+10] pour voir comment notre approche générique se comporte

par rapport à des méthodes spécifiques.

Enfin, une autre piste pour de recherches plus poussées sur la contrainte Order peut être trouvée dans les explications [Jus03]. En effet, les explications de la contrainte cumulative [Sch+09; Sch+11; SFS13] se sont avérées très efficaces pour résoudre les problèmes d'ordonnancement. Comme le filtrage sur d'autres variables de permutation que la variable actuelle est assez faible, voire inexistant, les explications pourraient être d'une grande aide pour filtrer d'autres variables de permutation que la variable actuelle, et différemment qu'à travers la contrainte allDifferent (ou bien allDiffPrec ou GeneralizedAllDiffPrec chaque fois que de telles contraintes peuvent être utiles).

La méthodologie développée pour la contrainte Order et ses spécifications pour résoudre différents problèmes rappelle que l'expressivité de la programmation par contraintes permet d'implémenter des techniques de résolution dédiées au sein des solveurs de contraintes.

En guise de conclusion, nous souhaitons attirer l'attention du lecteur sur l'hybridation de la programmation par contraintes avec d'autres techniques de résolution. En effet, la permissivité et l'expressivité de la programmation par contraintes permettent l'introduction de telles techniques au sein de propagateurs, permettant à la PPC de bénéficier de l'efficacité de telles techniques. Cela a conduit à de grandes réussites par le passé, comme avec les explications [Jus03] qui est une technique issue des solveurs SAT, ou avec la décomposition de Benders [Hoo07], ou bien avec les littéraux d'observation (*watch litterals* en anglais) [GJM06; Gen13] qui est un autre technique issue des solveurs SAT. Il semble donc être une bonne idée de s'inspirer des techniques de résolution issues d'autres domaines et outils d'optimisation combinatoire pour améliorer l'efficacité pratique des solveurs de contraintes.

DOCTORAT BRETAGNE LOIRE / MATHSTIC

IMT Atlantique
Bretagne-Pays de la Loire
École Mines-Télécom

Titre : Sur le tri de tâches pour résoudre des problèmes d'ordonnancement avec la programmation par contraintes

Mot clés : programmation par contraintes, ordonnancement, alldiffprec, liste ordonnée

Résumé : Au cours des deux dernières décennies, la programmation par contraintes s'est illustrée de par son efficacité à résoudre des problèmes d'ordonnancement. Grâce à la grande expressivité permise par le paradigme, différents algorithmes et techniques de résolution provenant d'autres domaines de l'Optimisation Combinatoire ont pu être intégrés au sein des solveurs de contraintes. Toutefois, cette grande expressivité fait que les solveurs ne sont pas des boîtes noires et demandent une expertise pour être paramétrés correctement pour résoudre efficacement les problèmes souhaités. Dans cette thèse, nous explorons l'introduction et l'utilisation d'algorithmes de liste ordonnée en programmation par contraintes pour résoudre des problèmes d'ordonnancement. Nous revisitons également la contrainte AllDiffPrec, définie comme une contrainte Alldifferent et des précédences entre variables, pour laquelle nous proposons également un nouvel algorithme de filtrage.

Title: On the use of tasks ordering to solve scheduling problems with constraint programming

Keywords: constraint programming, scheduling, ordering, alldiffprec, list ordering

Abstract: During the last two decades, Constraint Programming gets very good results to solve scheduling problems. Thanks to the great expressivity of the paradigm, different algorithms and solving techniques from other fields within Combinatorial Optimisation have been integrated into constraint solvers. However this great expressivity comes with a price: constraint solvers are not the black box one might think of and require expertise to be correctly configured to efficiently solve problems. In this thesis, we explore the introduction and the use of list ordering algorithms into Constraint Programming to solve scheduling problems. We also revisit the AllDiffPrec constraint, defined as an Alldifferent constraint with precedence between some variables, for which we propose a new filtering algorithm.